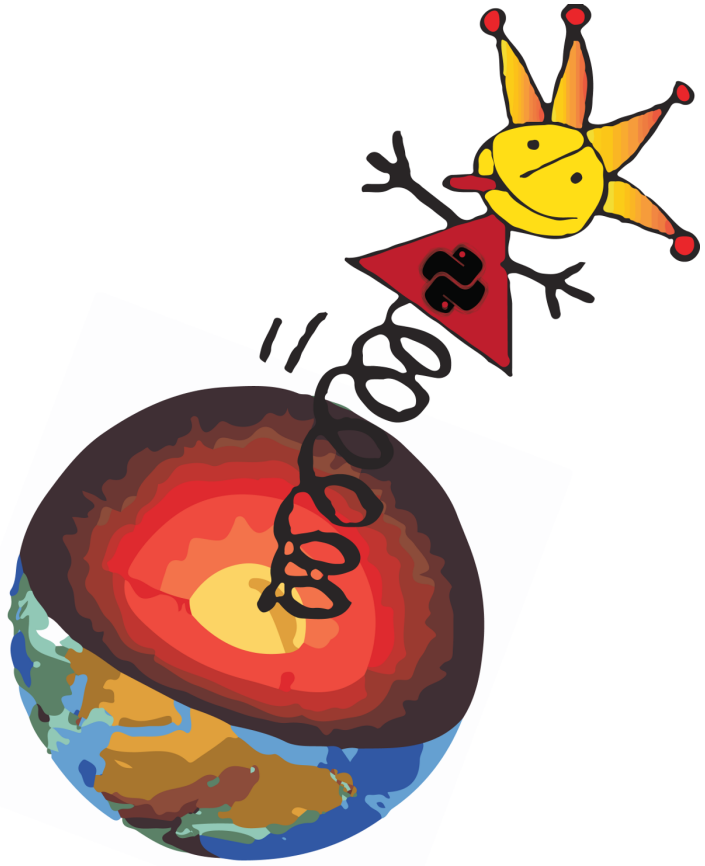
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

BurnMan

a thermodynamics and thermoelasticity toolkit

User Manual

Version 1.2.0a0



Robert Myhill
Sanne Cottaar
Timo Heister
Ian Rose
Cayman Unterborn

http://geodynamics.org

CONTENTS

CHAPTER

ONE

INTRODUCING BURNMAN 1.2

1.1 Overview

BurnMan is an open source mineral physics and seismological toolkit written in Python which can enable a user to calculate (or fit) the physical and chemical properties of endmember minerals, fluids/melts, solutions, and composite assemblages.

Properties which BurnMan can calculate include:

- the thermodynamic free energies, allowing phase equilibrium calculations, endmember activities, chemical potentials and oxygen (and other) fugacities.
- entropy, enabling the user to calculate isentropes for a given assemblage.
- volume, to allow the user to create density profiles.
- seismic velocities, including Voigt-Reuss-Hill and Hashin-Strikman bounds and averages.

The toolkit itself comes with a large set of classes and functions which are designed to allow the user to easily combine mineral physics with geophysics, and geodynamics. The features of BurnMan include:

- the full codebase, which includes implementations of many static and thermal equations of state (including Vinet, Birch Murnaghan, Mie-Debye-Grueneisen, Modified Tait), and solution models (ideal, symmetric, asymmetric, subregular).
- popular endmember and solution datasets already coded into burnman-usable format (including [HollandPowell11], [SLB05] and [SLB11])
- Optimal least squares fitting routines for multivariate data with (potentially correlated) errors in pressure and temperature. As an example, such functions can be used to simultaneously fit volumes, seismic velocities and enthalpies.
- a "Planet" class, which self-consistently calculates gravity profiles, mass, moment of inertia of planets given the chemical and temperature structure of a planet
- published geotherms
- a tutorial on the basic use of BurnMan
- a large collection of annotated examples
- a set of high-level functions which create files readable by seismological and geodynamic software, including: Mineos [MWF11], AxiSEM [NissenMeyervanDrielStahler+14] and ASPECT

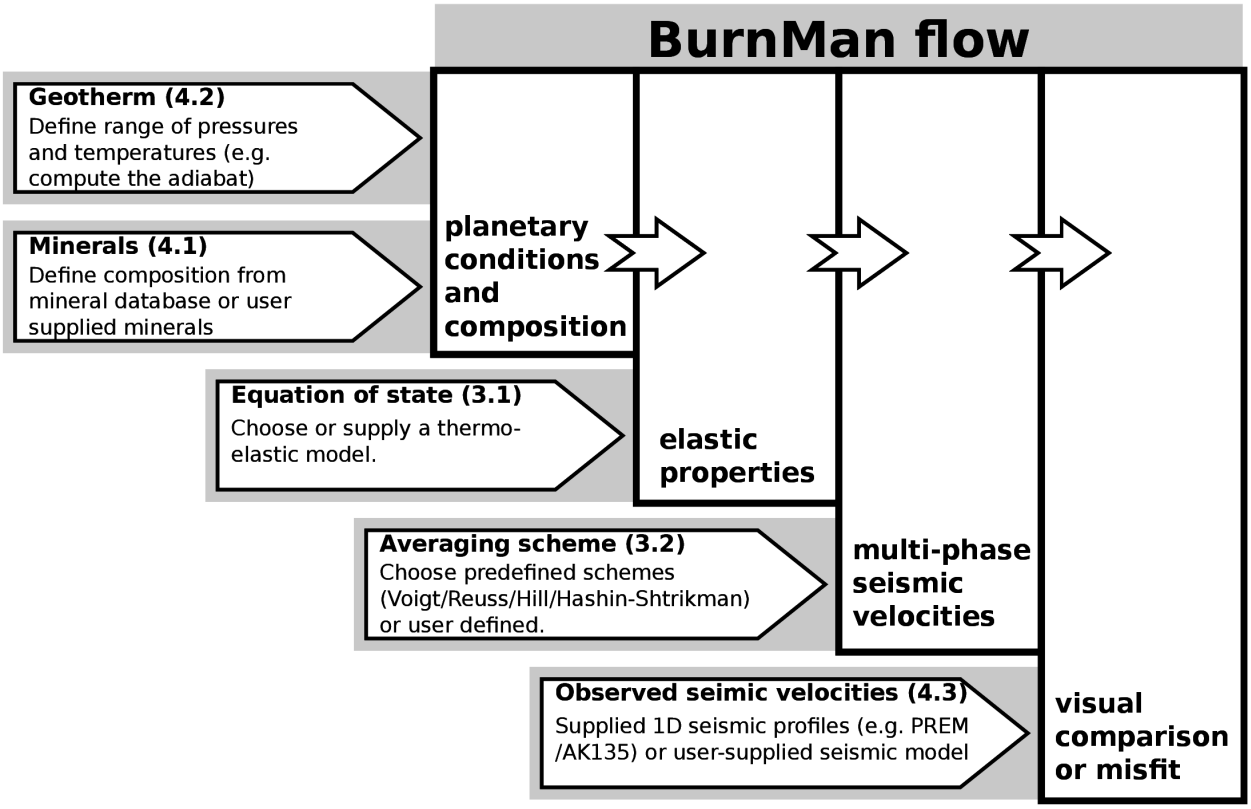
- an extensive suite of unit tests to ensure code functions as intended
- a series of benchmarks comparing BurnMan output with published data
- a directory containing user-contributed code from published papers

BurnMan makes extensive use of SciPy, NumPy and SymPy which are widely used Python libraries for scientific computation. Matplotlib is used to display results and produce publication quality figures. The computations are consistently formulated in terms of SI units.

The code documentation including class and function descriptions can be found online at http://burnman.readthedocs.io.

This software has been designed to allow the end-user a great deal of freedom to do whatever calculations they may wish and to add their own modules. The underlying Python classes have been designed to make new endmember, solution and composite models easy to read and create. We have endeavoured to provide examples and benchmarks which cover the most popular uses of the software, some of which are included in the figure below. This list is certainly not exhaustive, and we will definitely have missed interesting applications. We will be very happy to accept contributions in form of corrections, examples, or new features.

1.2 Structure



1.3 Requirements

- Python 3.8+
- Python modules: NumPy, SciPy, SymPy, Sparse, Matplotlib

1.3.1 Optional modules

Needed for some functionality:

- cvxpy: required for some least squares fitting routines and solution polytope calculations.
- pycddlib: required for solution polytope calculations.
- autograd: required for esoteric solution models defined using a single excess function. Not required for the vast majority of users.

1.4 Installation

Installation of BurnMan is mostly platform independent. As long as you know how to use a terminal, the process should be straightforward. The following instructions should help, but let us know if you have any problems.

1.4.1 Dependencies

First, make sure you have a sufficiently recent version of python installed on your machine (see above for the latest requirements). To check your version of python, type the following in a terminal:

python –version

If your version is not recent enough, visit https://www.python.org/ to find out how to install a newer version.

Once you have checked your version of python, you should make sure you have installed the python module pip. We will use this module to install BurnMan. If you don't have it already, you can install it by opening a terminal window and typing:

python -m ensurepip –upgrade

Mac users will also need to install Xcode, which can be found in the MacStore.

1.4.2 Stable version

If you are only interested in using BurnMan (rather than developing the software), and you aren't interested in any of the latest changes, you can install the stable version by typing the following into a terminal window:

```
python -m pip install burnman
```

This method of installation does not give you easy access to all the examples, or the test suite. These can be found in the latest release package which can be downloaded from https://github.com/geodynamics/burnman/releases.

1.4.3 Development version

If you want to install the development version of BurnMan (with all the latest features), you will first need to download the source code. The best way to do this is by using git (a version control system). To install git, follow the instructions at https://git-scm.com/downloads.

Then, using a terminal, navigate to the directory into which you want to clone the BurnMan repository, and type

```
git clone https://github.com/geodynamics/burnman.git
```

(If you don't want to use git, you can download the current master branch from https://github.com/geodynamics/burnman/archive/master.zip.)

Once the repository is cloned, navigate to the top-level directory by typing *cd burnman* in the terminal, and then install BurnMan, either in static mode: *python -m pip install .* or in development mode (if you want to develop or modify the code): *python -m pip install -e .*.

1.4.4 Checking that the installation worked

To check that the installation has worked, you can run the test suite (*./test.sh*). This takes a few minutes to run.

A more basic check that BurnMan is installed is to navigate to the Burnman examples directory and type:

```
python example_beginner.py
```

If figures show up, BurnMan has been installed.

1.5 Citing BurnMan

If you use BurnMan in your work, we ask that you cite the following publications:

- Myhill, R., Cottaar, S., Heister, T., Rose, I., and Unterborn, C. (2022): BurnMan v1.1.0 [Software]. Computational Infrastructure for Geodynamics. Zenodo. (https://doi.org/10.5281/zenodo.7080174)
- Cottaar S., Heister, T., Rose, I., and Unterborn, C., 2014, BurnMan: A lower mantle mineral physics toolkit, Geochemistry, Geophysics, and Geosystems, 15(4), 1164-1179 (https://doi.org/10.1002/2013GC005122)

1.6 Contributing to BurnMan

If you would like to contribute bug fixes, new functions or new modules to the existing codebase, please contact us at info@burnman.org or make a pull request at https://github.com/geodynamics/burnman.

BurnMan also includes a contrib directory that contains python and ipython scripts used to reproduce published results. We welcome the submission of new contributions to this directory. As with the contribution of code, please contact us at info@burnman.org or make a pull request at https://github.com/geodynamics/burnman.

1.7 Acknowledgement and Support

- This project was initiated at, and follow-up research support was received through, Cooperative Institute of Deep Earth Research, CIDER (NSF FESD grant 1135452) – see www.deep-earth.org
- We thank all the members of the CIDER Mg/Si team for their input: Valentina Magni, Yu Huang, JiaChao Liu, Marc Hirschmann, and Barbara Romanowicz. We also thank Lars Stixrude for providing benchmarking calculations and Zack Geballe, Motohiko Murakami, Bill McDonough, Quentin Williams, Wendy Panero, and Wolfgang Bangerth for helpful discussions.
- We thank CIG (www.geodynamics.org) for support and accepting our donation of BurnMan as an official project.

CHAPTER

TWO

MATHEMATICAL BACKGROUND

Here is a bit of background on the methods used to calculate thermoelastic and thermodynamic properties in BurnMan. More detail can be found in the cited papers.

2.1 Endmember Properties

2.1.1 Calculating Thermoelastic Properties

To calculate the bulk (K) modulus, shear modulus (G) and density (ρ) of a material at a given pressure (P) and temperature (T), optionally defined by a geotherm) and determine the seismic velocities (V_S, V_P, V_Φ), one uses an Equation of State (EoS). Currently the following EoSs are supported in BurnMan:

- Birch-Murnaghan finite-strain EoS (excludes temperature effects, [Poi91]),
- Birch-Murnaghan finite-strain EoS with a Mie-Grüneisen-Debye thermal correction, as formulated by [SLB05].
- Birch-Murnaghan finite-strain EoS with a Mie-Grüneisen-Debye thermal correction, as formulated by [MBR+07].
- Modified Tait EoS (excludes temperature effects, [HuangChow74]),
- Modified Tait EoS with a pseudo-Einstein model for thermal corrections, as formulated by [Holland-Powell11].
- Compensated-Redlich-Kwong for fluids, as formulated by [HP91].

To calculate these thermoelastic parameters, the EoS requires the user to input the pressure, temperature, and the phases and their molar fractions. These inputs and outputs are further discussed in *User input*.

2.1.1.1 Birch-Murnaghan (isothermal)

The Birch-Murnaghan equation is an isothermal Eulerian finite-strain EoS relating pressure and volume. The negative finite-strain (or compression) is defined as

$$f = \frac{1}{2}\left[\left(\frac{V}{V_0}\right)^{-2/3} - 1\right], \tag{2.1}$$

where V is the volume at a given pressure and V_0 is the volume at a reference state ($P = 10^5$ Pa , T = 300 K). The pressure and elastic moduli are derived from a third-order Taylor expansion of Helmholtz free energy in f and evaluating the appropriate volume and strain derivatives (e.g., [Poi91]). For an isotropic material one obtains for the pressure, isothermal bulk modulus, and shear modulus:

$$P = 3K_0 f\left(1+2f\right)^{5/2}\left[1 + \frac{3}{2}\left(K_0' - 4\right)f\right], \tag{2.2}$$

$$K_T = (1+2f)^{5/2}\Big[K_0 + (3K_0K_0' - 5K_0)f + \frac{27}{2}(K_0K_0' - 4K_0)f^2\Big], \tag{2.3}$$

$$G = (1+2f)^{5/2}\Big[G_0 + (3K_0G_0' - 5G_0)f + (6K_0G_0' - 24K_0 - 14G_0 + \frac{9}{2}K_0K_0')f^2\Big]. \tag{2.4}$$

Here K_0 and G_0 are the reference bulk modulus and shear modulus and K_0' and G_0' are the derivative of the respective moduli with respect to pressure.

BurnMan has the option to use the second-order expansion for shear modulus by dropping the f^2 terms in these equations (as is sometimes done for experimental fits or EoS modeling).

2.1.1.2 Modified Tait (isothermal)

The Modified Tait equation of state was developed by [HuangChow74]. It has the considerable benefit of allowing volume to be expressed as a function of pressure. It performs very well to pressures and temperatures relevant to the deep Earth [HollandPowell11].

$$\begin{aligned}\frac{V_{P,T}}{V_{1bar,298K}} &= 1 - a(1-(1+bP)^{-c}),\\ a &= \frac{1+K_0'}{1+K_0'+K_0K_0''},\\ b &= \frac{K_0'}{K_0} - \frac{K_0''}{1+K_0'},\\ c &= \frac{1+K_0'+K_0K_0''}{K_0'^2+K_0'-K_0K_0''}\end{aligned} \tag{2.5}$$

2.1.1.3 Mie-Grüneisen-Debye (thermal correction to Birch-Murnaghan)

The Debye model for the Helmholtz free energy can be written as follows [MBR+07]

$$\begin{aligned}
\mathcal{F} &= \frac{9nRT}{V}\frac{1}{x^3}\int_0^x \xi^2 \ln(1-e^{-\xi})d\xi, \\
x &= \theta/T, \\
\theta &= \theta_0 \exp\left(\frac{\gamma_0-\gamma}{q_0}\right), \\
\gamma &= \gamma_0\left(\frac{V}{V_0}\right)^{q_0}
\end{aligned}$$

where θ is the Debye temperature and γ is the Grüneisen parameter.

Using thermodynamic relations we can derive equations for the thermal pressure and bulk modulus

$$\begin{aligned}
P_{th}(V,T) &= -\frac{\partial \mathcal{F}(\mathcal{V},\mathcal{T})}{\partial V}, \\
&= \frac{3n\gamma RT}{V}D(x), \\
K_{th}(V,T) &= -V\frac{\partial P(V,T)}{\partial V}, \\
&= \frac{3n\gamma RT}{V}\gamma\left[(1-q_0-3\gamma)D(x)+3\gamma\frac{x}{e^x-1}\right], \\
D(x) &= \frac{3}{x^3}\int_0^x \frac{\xi^3}{e^\xi-1}d\xi
\end{aligned}$$

The thermal shear correction used in BurnMan was developed by [HamaSuito98]

$$G_{th}(V,T) = \frac{3}{5}\left[K_{th}(V,T) - 2\frac{3nRT}{V}\gamma D(x)\right]$$

The total pressure, bulk and shear moduli can be calculated from the following sums

$$\begin{aligned}
P(V,T) &= P_{\text{ref}}(V,T_0) + P_{th}(V,T) - P_{th}(V,T_0), \\
K(V,T) &= K_{\text{ref}}(V,T_0) + K_{th}(V,T) - K_{th}(V,T_0), \\
G(V,T) &= G_{\text{ref}}(V,T_0) + G_{th}(V,T) - G_{th}(V,T_0)
\end{aligned}$$

This equation of state is substantially the same as that in SLB2005 (see below). The primary differences are in the thermal correction to the shear modulus and in the volume dependences of the Debye temperature and the Gruneisen parameter.

2.1.1.4 HP2011 (thermal correction to Modified Tait)

The thermal pressure can be incorporated into the Modified Tait equation of state, replacing P with $P - (P_{\text{th}} - P_{\text{th0}})$ in Equation (2.5) [HollandPowell11]. Thermal pressure is calculated using a Mie-Grüneisen equation of state and an Einstein model for heat capacity, even though the Einstein model is not actually used

for the heat capacity when calculating the enthalpy and entropy (see following section).

$$P_{\text{th}} = \frac{\alpha_0 K_0 E_{\text{th}}}{C_{V0}},$$
$$E_{\text{th}} = 3nR\Theta \left(0.5 + \frac{1}{\exp(\frac{\Theta}{T}) - 1} \right),$$
$$C_V = 3nR \frac{(\frac{\Theta}{T})^2 \exp(\frac{\Theta}{T})}{(\exp(\frac{\Theta}{T}) - 1)^2}$$

Θ is the Einstein temperature of the crystal in Kelvin, approximated for a substance i with n_i atoms in the unit formula and a molar entropy S_i using the empirical formula

$$\Theta_i = \frac{10636}{S_i/n_i + 6.44}$$

2.1.1.5 SLB2005 (for solids, thermal)

Thermal corrections for pressure, and isothermal bulk modulus and shear modulus are derived from the Mie-Grüneisen-Debye EoS with the quasi-harmonic approximation. Here we adopt the formalism of [SLB05] where these corrections are added to equations (2.2)–(2.4):

$$\begin{aligned} P_{th}(V,T) &= \frac{\gamma \Delta \mathcal{U}}{V}, \\ K_{th}(V,T) &= (\gamma + 1 - q)\frac{\gamma \Delta \mathcal{U}}{V} - \gamma^2 \frac{\Delta(C_V T)}{V}, \\ G_{th}(V,T) &= -\frac{\eta_S \Delta \mathcal{U}}{V}. \end{aligned} \tag{2.6}$$

The Δ refers to the difference in the relevant quantity from the reference temperature (300 K). γ is the Grüneisen parameter, q is the logarithmic volume derivative of the Grüneisen parameter, η_S is the shear strain derivative of the Grüneisen parameter, C_V is the heat capacity at constant volume, and $\mathcal{U}$ is the internal energy at temperature T. C_V and $\mathcal{U}$ are calculated using the Debye model for vibrational energy of a lattice.

These quantities are calculated as follows:

$$
\begin{aligned}
C_V &= 9nR\left(\frac{T}{\theta}\right)^3 \int_0^{\frac{\theta}{T}} \frac{e^\tau \tau^4}{(e^\tau - 1)^2} d\tau, \\
\mathcal{U} &= 9nRT\left(\frac{T}{\theta}\right)^3 \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau - 1)} d\tau, \\
\gamma &= \frac{1}{6}\frac{\nu_0^2}{\nu^2}(2f+1)\left[a_{ii}^{(1)} + a_{iikk}^{(2)} f\right], \\
q &= \frac{1}{9\gamma}\left[18\gamma^2 - 6\gamma - \frac{1}{2}\frac{\nu_0^2}{\nu^2}(2f+1)^2 a_{iikk}^{(2)}\right], \\
\eta_S &= -\gamma - \frac{1}{2}\frac{\nu_0^2}{\nu^2}(2f+1)^2 a_S^{(2)}, \\
\frac{\nu^2}{\nu_0^2} &= 1 + a_{ii}^{(1)} f + \frac{1}{2} a_{iikk}^{(2)} f^2, \\
a_{ii}^{(1)} &= 6\gamma_0, \\
a_{iikk}^{(2)} &= -12\gamma_0 + 36\gamma_0^2 - 18 q_0 \gamma_0, \\
a_S^{(2)} &= -2\gamma_0 - 2\eta_{S0},
\end{aligned}
$$

where θ is the Debye temperature of the mineral, ν is the frequency of vibrational modes for the mineral, n is the number of atoms per formula unit (e.g. 2 for periclase, 5 for perovskite), and R is the gas constant. Under the approximation that the vibrational frequencies behave the same under strain, we may identify $\nu/\nu_0 = \theta/\theta_0$. The quantities γ_0, η_{S0} q_0, and θ_0 are the experimentally determined values for those parameters at the reference state.

Due to the fact that a planetary mantle is rarely isothermal along a geotherm, it is more appropriate to use the adiabatic bulk modulus K_S instead of K_T, which is calculated using

$$K_S = K_T(1 + \gamma\alpha T), \tag{2.7}$$

where α is the coefficient of thermal expansion:

$$\alpha = \frac{\gamma C_V V}{K_T}. \tag{2.8}$$

There is no difference between the isothermal and adiabatic shear moduli for an isotropic solid. All together this makes an eleven parameter EoS model, which is summarized in the Table below. For more details on the EoS, we refer readers to [SLB05].

User Input	Symbol	Definition	Units
V_0	V_0	**Volume at P =** 10^5 Pa , T = 300 K	m ^{3} mol $^{-1}$
K_0	K_0	Isothermal bulk modulus at *P=10^5* Pa, T = 300 K	Pa
Kprime_0	K'_0	Pressure derivative of K_0	
G_0	G_0	Shear modulus at P = 10^5 Pa, T = 300 K	Pa
Gprime_0	G'_0	Pressure derivative of G_0	
molar_mass	μ	mass per mole formula unit	kg mol^{-1}
n	n	number of atoms per formula unit	
Debye_0	θ_0	Debye Temperature	K
grueneisen_0	γ_0	Grüneisen parameter at P = 10^5 Pa, T = 300 K	
q0	q_0	Logarithmic volume derivative of the Grüneisen parameter	
eta_s_0	η_{S0}	Shear strain derivative of the Grüneisen parameter	

This equation of state is substantially the same as that of the Mie-Gruneisen-Debye (see above). The primary differences are in the thermal correction to the shear modulus and in the volume dependences of the Debye temperature and the Gruneisen parameter.

2.1.1.6 Compensated-Redlich-Kwong (for fluids, thermal)

The CORK equation of state [HP91] is a simple virial-type extension to the modified Redlich-Kwong (MRK) equation of state. It was designed to compensate for the tendency of the MRK equation of state to overestimate volumes at high pressures and accommodate the volume behaviour of coexisting gas and liquid phases along the saturation curve.

$$
\begin{aligned}
V &= \frac{RT}{P} + c_1 - \frac{c_0 RT^{0.5}}{(RT + c_1 P)(RT + 2c_1 P)} + c_2 P^{0.5} + c_3 P, \\
c_0 &= c_{0,0} T_c^{2.5}/P_c + c_{0,1} T_c^{1.5}/P_c T, \\
c_1 &= c_{1,0} T_c/P_c, \\
c_2 &= c_{2,0} T_c/P_c^{1.5} + c_{2,1}/P_c^{1.5} T, \\
c_3 &= c_{3,0} T_c/P_c^2 + c_{3,1}/P_c^2 T
\end{aligned}
$$

2.1.2 Calculating Thermodynamic Properties

So far, we have concentrated on the thermoelastic properties of minerals. There are, however, additional thermodynamic properties which are required to describe the thermal properties such as the energy, entropy and heat capacity. These properties are related by the following expressions:

$$\mathcal{G} = \mathcal{E} - T\mathcal{S} + PV = \mathcal{H} - TS = \mathcal{F} + PV \tag{2.9}$$

where P is the pressure, T is the temperature and $\mathcal{E}$, $\mathcal{F}$, $\mathcal{H}$, $\mathcal{S}$ and V are the molar internal energy, Helmholtz free energy, enthalpy, entropy and volume respectively.

2.1.2.1 HP2011

$$\begin{aligned}
\mathcal{G}(P,T) &= \mathcal{H}_{\text{1 bar, T}} - T\mathcal{S}_{\text{1 bar, T}} + \int_{\text{1 bar}}^{P} V(P,T)\, dP, \\
\mathcal{H}_{\text{1 bar, T}} &= \Delta_f \mathcal{H}_{\text{1 bar, 298 K}} + \int_{298}^{T} C_P\, dT, \\
\mathcal{S}_{\text{1 bar, T}} &= \mathcal{S}_{\text{1 bar, 298 K}} + \int_{298}^{T} \frac{C_P}{T}\, dT, \\
\int_{\text{1 bar}}^{P} V(P,T)\, dP &= PV_0 \left(1 - a + \left(a \frac{(1-bP_{th})^{1-c} - (1+b(P-P_{th}))^{1-c}}{b(c-1)P}\right)\right)
\end{aligned} \tag{2.10}$$

The heat capacity at one bar is given by an empirical polynomial fit to experimental data

$$C_p = a + bT + cT^{-2} + dT^{-0.5}$$

The entropy at high pressure and temperature can be calculated by differentiating the expression for $\mathcal{G}$ with respect to temperature

$$\begin{aligned}
\mathcal{S}(P,T) &= S_{\text{1 bar, T}} + \frac{\partial \int V dP}{\partial T}, \\
\frac{\partial \int V dP}{\partial T} &= V_0 \alpha_0 K_0 a \frac{C_{V0}(T)}{C_{V0}(T_{\text{ref}})} ((1+b(P-P_{th}))^{-c} - (1-bP_{th})^{-c})
\end{aligned}$$

Finally, the enthalpy at high pressure and temperature can be calculated

$$\mathcal{H}(P,T) = \mathcal{G}(P,T) + T\mathcal{S}(P,T)$$

2.1.2.2 SLB2005

The Debye model yields the Helmholtz free energy and entropy due to lattice vibrations

$$\begin{aligned}
\mathcal{G} &= \mathcal{F} + PV, \\
\mathcal{F} &= nRT \left(3\ln(1 - e^{-\frac{\theta}{T}}) - \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau - 1)} d\tau\right), \\
\mathcal{S} &= nR \left(4 \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau - 1)} d\tau - 3\ln(1 - e^{-\frac{\theta}{T}})\right), \\
\mathcal{H} &= \mathcal{G} + T\mathcal{S}
\end{aligned}$$

2.1.3 Property modifiers

The thermodynamic models above consider the effects of strain and quasiharmonic lattice vibrations on the free energies of minerals at given temperatures and pressures. There are a number of additional processes, such as isochemical order-disorder and magnetic effects which also contribute to the total free energy of a phase. Corrections for these additional processes can be applied in a number of different ways. Burnman currently includes implementations of the following:

- Linear excesses (useful for DQF modifications for [HollandPowell11])
- Tricritical Landau model (two formulations)
- Bragg-Williams model
- Magnetic excesses

In all cases, the excess Gibbs free energy $\mathcal{G}$ and first and second partial derivatives with respect to pressure and temperature are calculated. The thermodynamic properties of each phase are then modified in a consistent manner; specifically:

$$
\begin{aligned}
\mathcal{G} &= \mathcal{G}_o + \mathcal{G}_m, \\
\mathcal{S} &= \mathcal{S}_o - \frac{\partial \mathcal{G}}{\partial T}_m, \\
\mathcal{V} &= \mathcal{V}_o + \frac{\partial \mathcal{G}}{\partial P}_m, \\
K_T &= \mathcal{V} / \left(\frac{\mathcal{V}_o}{K_T o} - \frac{\partial^2 \mathcal{G}}{\partial P^2} \right)_m, \\
C_p &= C_p o - T \frac{\partial^2 \mathcal{G}}{\partial T^2}_m, \\
\alpha &= \left(\alpha_o \mathcal{V}_o + \frac{\partial^2 \mathcal{G}}{\partial P \partial T}_m \right) / \mathcal{V}, \\
\mathcal{H} &= \mathcal{G} + T\mathcal{S}, \\
\mathcal{F} &= \mathcal{G} - P\mathcal{V}, \\
C_v &= C_p - \mathcal{V} T \alpha^2 K_T, \\
\gamma &= \frac{\alpha K_T \mathcal{V}}{C_v}, \\
K_S &= K_T \frac{C_p}{C_v}
\end{aligned}
$$

Subscripts $_o$ and $_m$ indicate original properties and modifiers respectively. Importantly, this allows us to stack modifications such as multiple Landau transitions in a simple and straightforward manner. In the burnman code, we add property modifiers as an attribute to each mineral as a list. For example:

```
from burnman.minerals import SLB_2011
stv = SLB_2011.stishovite()
stv.property_modifiers = [
['landau',
{'Tc_0': -4250.0, 'S_D': 0.012, 'V_D': 1e-09}]]
```

(continues on next page)

(continued from previous page)

```
['linear',
{'delta_E': 1.e3, 'delta_S': 0., 'delta_V': 0.}]]
```

Each modifier is a list with two elements, first the name of the modifier type, and second a dictionary with the required parameters for that model. A list of parameters for each model is given in the following sections.

2.1.3.1 Linear excesses (linear)

A simple linear correction in pressure and temperature. Parameters are 'delta_E', 'delta_S' and 'delta_V'.

$$\begin{aligned}
\mathcal{G} = \Delta\mathcal{E} - T\Delta\mathcal{S} + P\Delta\mathcal{V}, \\
\frac{\partial \mathcal{G}}{\partial T} = -\Delta\mathcal{S}, \\
\frac{\partial \mathcal{G}}{\partial P} = \Delta\mathcal{V}, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} = 0, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} = 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} = 0
\end{aligned}$$

2.1.3.2 Tricritical Landau model (landau)

Applies a tricritical Landau correction to the properties of an endmember which undergoes a displacive phase transition. These transitions are not associated with an activation energy, and therefore they occur rapidly compared with seismic wave propagation. Parameters are 'Tc_0', 'S_D' and 'V_D'.

This correction follows [Putnis92], and is done relative to the completely *ordered* state (at 0 K). It therefore differs in implementation from both [SLB11] and [HollandPowell11], who compute properties relative to the completely disordered state and standard states respectively. The current implementation is preferred, as the excess entropy (and heat capacity) terms are equal to zero at 0 K.

$$Tc = Tc_0 + \frac{V_D P}{S_D}$$

If the temperature is above the critical temperature, Q (the order parameter) is equal to zero, and the Gibbs free energy is simply that of the disordered phase:

$$\begin{aligned}
\mathcal{G}_{\text{dis}} = -S_D \left((T - Tc) + \frac{Tc_0}{3} \right), \\
\frac{\partial \mathcal{G}}{\partial P}_{\text{dis}} = V_D, \\
\frac{\partial \mathcal{G}}{\partial T}_{\text{dis}} = -S_D
\end{aligned}$$

If temperature is below the critical temperature, Q is between 0 and 1. The gibbs free energy can be described thus:

$$Q^2 = \sqrt{\left(1 - \frac{T}{Tc}\right)},$$
$$\mathcal{G} = S_D \left((T - Tc)Q^2 + \frac{Tc_0 Q^6}{3}\right) + \mathcal{G}_{\text{dis}},$$
$$\frac{\partial \mathcal{G}}{\partial P} = -V_D Q^2 \left(1 + \frac{T}{2Tc}\left(1. - \frac{Tc_0}{Tc}\right)\right) + \frac{\partial \mathcal{G}}{\partial P}_{\text{dis}},$$
$$\frac{\partial \mathcal{G}}{\partial T} = S_D Q^2 \left(\frac{3}{2} - \frac{Tc_0}{2Tc}\right) + \frac{\partial \mathcal{G}}{\partial T}_{\text{dis}},$$
$$\frac{\partial^2 \mathcal{G}}{\partial P^2} = V_D^2 \frac{T}{S_D Tc^2 Q^2}\left(\frac{T}{4Tc}\left(1. + \frac{Tc_0}{Tc}\right) + Q^4\left(1. - \frac{Tc_0}{Tc}\right) - 1\right),$$
$$\frac{\partial^2 \mathcal{G}}{\partial T^2} = -\frac{S_D}{TcQ^2}\left(\frac{3}{4} - \frac{Tc_0}{4Tc}\right),$$
$$\frac{\partial^2 \mathcal{G}}{\partial P \partial T} = \frac{V_D}{2TcQ^2}\left(1 + \left(\frac{T}{2Tc} - Q^4\right)\left(1 - \frac{Tc_0}{Tc}\right)\right)$$

2.1.3.3 Tricritical Landau model (landau_hp)

Applies a tricritical Landau correction similar to that described above. However, this implementation follows [HollandPowell11], who compute properties relative to the standard state. Parameters are 'P_0', 'T_0', 'Tc_0', 'S_D' and 'V_D'.

It is worth noting that the correction described by [HollandPowell11] has been incorrectly used throughout the geological literature, particularly in studies involving magnetite (which includes studies comparing oxygen fugacities to the FMQ buffer (due to an incorrect calculation of the properties of magnetite). Note that even if the implementation is correct, it still allows the order parameter Q to be greater than one, which is physically impossible.

We include this implementation in order to reproduce the dataset of [HollandPowell11]. If you are creating your own minerals, we recommend using the standard implementation.

$$Tc = Tc0 + \frac{V_D P}{S_D}$$

If the temperature is above the critical temperature, Q (the order parameter) is equal to zero. Otherwise

$$Q^2 = \sqrt{\left(\frac{Tc - T}{Tc0}\right)}$$

$$\mathcal{G} = Tc_0 S_D \left(Q_0^2 - \frac{Q_0^6}{3}\right) - S_D \left(TcQ^2 - Tc_0 \frac{Q^6}{3}\right) - TS_D \left(Q_0^2 - Q^2\right) + PV_D Q_0^2,$$
$$\frac{\partial \mathcal{G}}{\partial P} = -V_D \left(Q^2 - Q_0^2\right),$$
$$\frac{\partial \mathcal{G}}{\partial T} = S_D \left(Q^2 - Q_0^2\right),$$

The second derivatives of the Gibbs free energy are only non-zero if the order parameter exceeds zero. Then

$$\frac{\partial^2 \mathcal{G}}{\partial P^2} = -\frac{V_D^2}{2S_D T c_0 Q^2},$$
$$\frac{\partial^2 \mathcal{G}}{\partial T^2} = -\frac{S_D}{2T c_0 Q^2},$$
$$\frac{\partial^2 \mathcal{G}}{\partial P \partial T} = \frac{V_D}{2T c_0 Q^2}$$

2.1.3.4 Bragg-Williams model (bragg_williams)

The Bragg-Williams model is a symmetric solution model between endmembers with an excess configurational entropy term determined by the specifics of order-disorder in the mineral, multiplied by some empirical factor. Expressions for the excess Gibbs free energy can be found in [HP96]. Parameters are 'deltaH', 'deltaV', 'Wh', 'Wv', 'n' and 'factor'.

2.1.3.5 Magnetic model (magnetic_chs)

This model approximates the excess energy due to magnetic ordering. It was originally described in [CHS87]. The expressions used by BurnMan can be found in [Sun91]. Parameters are 'structural_parameter', 'curie_temperature'[2] (zero pressure value and pressure dependence) and 'magnetic_moment'[2] (zero pressure value and pressure dependence).

2.2 Calculating Solution Properties

Many phases (whether minerals, melts, fluids or gases) can exist over a finite region of composition space. These spaces are bounded by endmembers (which may themselves not be stable), and each phase can then be described as a solution of those endmembers. In a solid solution, different elements substitute for one another on distinct crystallographic sites in the structure. For example, low pressure silicate garnets have two distinct sites on which mixing takes place; a dodecahedral site (of which there are three per unit cell on an eight-cation basis) and octahedral site (of which there are two per unit cell). A third tetrahedral cation site (three per unit cell) is usually assumed to be occupied solely by silicon, and therefore can be ignored in solution calculations. The chemical formula of many low pressure garnets exist within the solution:

$$[\mathrm{Mg,Fe,Mn,Ca}]_3[\mathrm{Al,Fe,Cr}]_2\mathrm{Si}_3\mathrm{O}_{12}$$

We typically calculate solution properties by appropriate differentiation of the Gibbs energy, where

$$\mathcal{G} = \sum_i n_i \left(\mathcal{G}_i + RT \ln \alpha_i\right)$$

$$\alpha_i = \gamma_i \alpha_{\text{ideal},i}$$

2.2.1 Implemented models

2.2.1.1 Ideal solutions

A solution is not simply a mechanical mixture of its constituent endmembers. The mixing of different species results in an excess configurational entropy S. In Bragg-Williams-type solutions, the entropy only depends on the amounts of species on sites, and the site multiplicities.

$$\mathcal{S}_{\text{conf}} = Rx_c^s \ln \frac{x_c^s}{\sum_{c^s} x_c^s}$$

where s is a site in the lattice, c are the species mixing on site s. x_c^s is the absolute number of species c on site s in the lattice; it is calculated by multiplying the proportion of the species on the site by the multiplicity of the site per formula unit and the number of moles of formula units.

Solutions where this configurational entropy is the only deviation from a mechanical mixture are termed *ideal*, because the enthalpy of mixing is zero. In BurnMan, the multiplicities of each site are allowed to vary linearly between endmembers. This is known as a Temkin model [Tem45].

2.2.1.2 Symmetric solutions

Many real phases are not well approximated as ideal solutions. Deviations are the result of elastic and chemical interactions between ions with different physical and chemical characteristics. Regular (symmetric) solution models account for the simplest form of deviations from ideality by incorporating terms describing excess enthalpies, non-configurational entropies and volumes relative to the ideal solution model. These excess terms have the matrix form [DPWH07]

$$\mathcal{G}_{\text{excess}} = RT \ln \gamma = p^T W p$$

where p is a vector of molar fractions of each of the n endmembers and W is a strictly upper-triangular matrix of interaction terms between endmembers. Excesses within binary systems (i-j) have a quadratic form and a maximum of $W_{ij}/4$ half-way between the two endmembers.

2.2.1.3 Asymmetric solutions

Some solutions exhibit asymmetric excess terms. These can be accounted for with an asymmetric solution [DPWH07]

$$\mathcal{G}_{\text{excess}} = \alpha^T p(\phi^T W \phi)$$

α is a vector of "van Laar parameters" governing asymmetry in the excess properties.

$$\phi_i = \frac{\alpha_i p_i}{\sum_{k=1}^{n} \alpha_k p_k},$$
$$W_{ij} = \frac{2w_{ij}}{\alpha_i + \alpha_j} \text{for i<j}$$

The w_{ij} terms are a set of interaction terms between endmembers i and j. If all the α terms are equal to unity, a non-zero w yields an excess with a quadratic form and a maximum of $w/4$ half-way between the two endmembers.

2.2.1.4 Subregular solutions

An alternative way to create asymmetric solution models is to expand each binary term as a cubic expression [HW89]. In this case,

$$\mathcal{G}_{\text{excess}} = \sum_i \sum_{j>i} (p_i p_j^2 W_{ij} + p_j p_i^2 W_{ji} + \sum_{k>j>i} p_i p_j p_k W_{ijk})$$

Note the similarity with the symmetric solution model, the primary difference being that there are two interaction terms for each binary and also additional ternary terms.

2.2.1.5 Arbitrary solutions

BurnMan also allows the user to define their own excess Gibbs energy function for a solution model.

2.2.2 Thermodynamic and thermoelastic properties

From the preceeding equations, we can define the thermodynamic potentials of solutions:

$$\begin{aligned}
\mathcal{G}_{\text{SS}} &= \sum_i n_i \left(\mathcal{G}_i + RT \ln \alpha_i\right) \\
\mathcal{S}_{\text{SS}} &= \sum_i n_i \mathcal{S}_i + \mathcal{S}_{\text{conf}} - \frac{\partial \mathcal{G}_{\text{excess}}}{\partial T} \\
\mathcal{H}_{\text{SS}} &= \mathcal{G}_{\text{SS}} + T\mathcal{S}_{\text{SS}} \\
V_{\text{SS}} &= \sum_i n_i V_i + \frac{\partial \mathcal{G}_{\text{excess}}}{\partial P}
\end{aligned}$$

We can also define the derivatives of volume with respect to pressure and temperature

$$\begin{aligned}
\alpha_{P,\text{SS}} &= \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P = \left(\frac{1}{V_{\text{SS}}}\right)\left(\sum_i \left(n_i\, \alpha_i\, V_i\right)\right) \\
K_{T,\text{SS}} &= V\left(\frac{\partial P}{\partial V}\right)_T = V_{\text{SS}}\left(\frac{1}{\sum_i \left(n_i \frac{V_i}{K_{Ti}}\right)} + \frac{\partial P}{\partial V_{\text{excess}}}\right)
\end{aligned}$$

Making the approximation that the excess entropy has no temperature dependence

$$\begin{aligned}
C_{P,\text{SS}} &= \sum_i n_i C_{Pi} \\
C_{V,\text{SS}} &= C_{P,\text{SS}} - V_{\text{SS}}\, T\, \alpha_{\text{SS}}^2\, K_{T,\text{SS}} \\
K_{S,\text{SS}} &= K_{T,\text{SS}}\, \frac{C_{P,\text{SS}}}{C_{V,\text{SS}}} \\
\gamma_{\text{SS}} &= \frac{\alpha_{\text{SS}}\, K_{T,\text{SS}}\, V_{\text{SS}}}{C_{V,\text{SS}}}
\end{aligned}$$

2.2.3 Including order-disorder

Order-disorder can be treated trivially with solutions. The only difference between mixing between ordered and disordered endmembers is that disordered endmembers have a non-zero configurational entropy, which must be accounted for when calculating the excess entropy within a solution.

2.2.4 Including spin transitions

The regular solution formalism should provide an elegant way to model spin transitions in phases such as periclase and bridgmanite. High and low spin iron can be treated as different elements, providing distinct endmembers and an excess configurational entropy. Further excess terms can be added as necessary.

2.3 Calculating Multi-phase Composite Properties

2.3.1 Averaging schemes

After the thermoelastic parameters (K_S, G, ρ) of each phase are determined at each pressure and/or temperature step, these values must be combined to determine the seismic velocity of a multiphase assemblage. We define the volume fraction of the individual minerals in an assemblage:

$$\nu_i = n_i \frac{V_i}{V},$$

where V_i and n_i are the molar volume and the molar fractions of the i th individual phase, and V is the total molar volume of the assemblage:

$$V = \sum_i n_i V_i. \tag{2.11}$$

The density of the multiphase assemblage is then

$$\rho = \sum_i \nu_i \rho_i = \frac{1}{V} \sum_i n_i \mu_i, \tag{2.12}$$

where ρ_i is the density and μ_i is the molar mass of the i th phase.

Unlike density and volume, there is no straightforward way to average the bulk and shear moduli of a multiphase rock, as it depends on the specific distribution and orientation of the constituent minerals. BurnMan allows several schemes for averaging the elastic moduli: the Voigt and Reuss bounds, the Hashin-Shtrikman bounds, the Voigt-Reuss-Hill average, and the Hashin-Shtrikman average [WDOConnell76].

The Voigt average, assuming constant strain across all phases, is defined as

$$X_V = \sum_i \nu_i X_i, \tag{2.13}$$

where X_i is the bulk or shear modulus for the i th phase. The Reuss average, assuming constant stress across all phases, is defined as

$$X_R = \left(\sum_i \frac{\nu_i}{X_i} \right)^{-1}. \tag{2.14}$$

The Voigt-Reuss-Hill average is the arithmetic mean of Voigt and Reuss bounds:

$$X_{VRH} = \frac{1}{2}\left(X_V + X_R\right). \tag{2.15}$$

The Hashin-Shtrikman bounds make an additional assumption that the distribution of the phases is statistically isotropic and are usually much narrower than the Voigt and Reuss bounds [WDOConnell76]. This may be a poor assumption in regions of Earth with high anisotropy, such as the lowermost mantle, however these bounds are more physically motivated than the commonly-used Voigt-Reuss-Hill average. In most instances, the Voigt-Reuss-Hill average and the arithmetic mean of the Hashin-Shtrikman bounds are quite similar with the pure arithmetic mean (linear averaging) being well outside of both.

It is worth noting that each of the above bounding methods are derived from mechanical models of a linear elastic composite. It is thus only appropriate to apply them to elastic moduli, and not to other thermoelastic properties, such as wave speeds or density.

2.3.2 Computing seismic velocities

Once the moduli for the multiphase assemblage are computed, the compressional (P), shear (S) and bulk sound (Φ) velocities are then result from the equations:

$$V_P = \sqrt{\frac{K_S + \frac{4}{3}G}{\rho}}, \qquad V_S = \sqrt{\frac{G}{\rho}}, \qquad V_\Phi = \sqrt{\frac{K_S}{\rho}}. \tag{2.16}$$

To correctly compare to observed seismic velocities one needs to correct for the frequency sensitivity of attenuation. Moduli parameters are obtained from experiments that are done at high frequencies (MHz-GHz) compared to seismic frequencies (mHz-Hz). The frequency sensitivity of attenuation causes slightly lower velocities for seismic waves than they would be for high frequency waves. In BurnMan one can correct the calculated acoustic velocity values to those for long period seismic tomography [MA81]:

$$V_{S/P} = V_{S/P}^{\text{uncorr.}}\left(1 - \frac{1}{2}\cot(\frac{\beta\pi}{2})\frac{1}{Q_{S/P}}(\omega)\right).$$

Similar to [MBR+07], we use a β value of 0.3, which falls in the range of values of 0.2 to 0.4 proposed for the lower mantle (e.g. [KS90]). The correction is implemented for Q values of PREM for the lower mantle. As Q_S is smaller than Q_P, the correction is more significant for S waves. In both cases, though, the correction is minor compared to, for example, uncertainties in the temperature (corrections) and mineral physical parameters. More involved models of relaxation mechanisms can be implemented, but lead to the inclusion of more poorly constrained parameters, [MB07]. While attenuation can be ignored in many applications [TVV01], it might play a significant role in explaining strong variations in seismic velocities in the lowermost mantle [DGD+12].

2.4 Thermodynamic Equilibration

For a composite with fixed phases at a given pressure, temperature and composition, equilibrium is reached when the following relationships are satisfied:

$$0_i = R_{ij}\mu_j$$

where μ_j are the chemical potentials of all of the endmembers in all of the phases, and R_{ij} is an independent set of balanced reactions between endmembers.

It is generally true that at a fixed composition, one can choose two equilibrium constraints (such as fixed temperature, pressure, entropy, volume, phase proportion or some composition constraint) and solve for the remaining unknowns. In BurnMan, this can be achieved using the equilibrate function (see *Equilibrium Thermodynamics*).

2.5 User input

2.5.1 Mineralogical composition

A number of pre-defined minerals are included in the mineral library and users can create their own. The library includes wrapper functions to include a transition from the high-spin mineral to the low-spin mineral [LSMM13] or to combine minerals for a given iron number.

Standard minerals – The 'standard' mineral format includes a list of parameters given in the above table. Each mineral includes a suggested EoS with which the mineral parameters are derived. For some minerals the parameters for the thermal corrections are not yet measured or calculated, and therefore the corrections can not be applied. An occasional mineral will not have a measured or calculated shear moduli, and therefore can only be used to compute densities and bulk sound velocities. The mineral library is subdivided by citation. BurnMan includes the option to produce a LaTeX; table of the mineral parameters used. BurnMan can be easily setup to incorporate uncertainties for these parameters.

Minerals with a spin transition – A standard mineral for the high spin and low spin must be defined separately. These minerals are "wrapped," so as to switch from the high spin to high spin mineral at a give pressure. While not realistic, for the sake of simplicity, the spin transitions are considered to be sharp at a given pressure.

Minerals depending on Fe partitioning – The wrapper function can partition iron, for example between ferropericlase, fp, and perovskite, pv. It requires the input of the iron mol fraction with regards to Mg, X_{fp} and X_{pv}, which then defines the chemistry of an Mg-Fe solid solution according to $(\mathrm{Mg}_{1-X^{\mathrm{fp}}_{\mathrm{Fe}}}, \mathrm{Fe}_{X^{\mathrm{fp}}_{\mathrm{Fe}}})\mathrm{O}$ or $(\mathrm{Mg}_{1-X^{\mathrm{pv}}_{\mathrm{Fe}}}, \mathrm{Fe}_{X^{\mathrm{pv}}_{\mathrm{Fe}}})\mathrm{SiO}_3$. The iron mol fractions can be set to be constant or varying with P and T as needed. Alternatively one can calculate the iron mol fraction from the distribution coefficient K_D defined as

$$K_D = \frac{X^{\mathrm{pv}}_{\mathrm{Fe}}/X^{\mathrm{pv}}_{\mathrm{Mg}}}{X^{\mathrm{fp}}_{\mathrm{Fe}}/X^{\mathrm{fp}}_{\mathrm{Mg}}}. \tag{2.17}$$

We adopt the formalism of [NFR12] choosing a reference distribution coefficient K_{D0} and standard state volume change (Δv^0) for the Fe-Mg exchange between perovskite and ferropericlase

$$K_D = K_{D0} \exp\left(\frac{(P_0 - P)\Delta v^0}{RT}\right), \tag{2.18}$$

where R is the gas constant and P_0 the reference pressure. As a default, we adopt the average Δv^0 of [NFR12] of $2 \cdot 10^{-7}\ m^3 mol^{-1}$ and suggest using their K_{D0} value of 0.5.

The multiphase mixture of these minerals can be built by the user in three ways:

1. Molar fractions of an arbitrary number of pre-defined minerals, for example mixing standard minerals mg_perovskite ($\mathrm{MgSiO_3}$), fe_perovskite ($\mathrm{FeSiO_3}$), periclase (MgO) and wüstite (FeO).

2. A two-phase mixture with constant or (P, T) varying Fe partitioning using the minerals that include Fe-dependency, for example mixing $(\mathrm{Mg, Fe})\mathrm{SiO_3}$ and $(\mathrm{Mg, Fe})\mathrm{O}$ with a pre-defined distribution coefficient.

3. Weight percents (wt%) of (Mg, Si, Fe) and distribution coefficient (includes (P,T)-dependent Fe partitioning). This calculation assumes that each element is completely oxidized into its corresponding oxide mineral (MgO, FeO, $\mathrm{SiO_2}$) and then combined to form iron-bearing perovskite and ferropericlase taking into account some Fe partition coefficient.

2.5.2 Geotherm

Unlike the pressure, the temperature of the lower mantle is relatively unconstrained. As elsewhere, BurnMan provides a number of built-in geotherms, as well as the ability to use user-defined temperature-depth relationships. A geotherm in BurnMan is an object that returns temperature as a function of pressure. Alternatively, the user could ignore the geothermal and compute elastic velocities for a range of temperatures at any give pressure.

Currently, we include geotherms published by [BS81] and [And82]. Alternatively one can use an adiabatic gradient defined by the thermoelastic properties of a given mineralogical model. For a homogeneous material, the adiabatic temperature profile is given by integrating the ordinary differential equation (ODE)

$$\left(\frac{\mathrm{d}T}{\mathrm{d}P}\right)_S = \frac{\gamma T}{K_S}. \tag{2.19}$$

This equation can be extended to multiphase composite using the first law of thermodynamics to arrive at

$$\left(\frac{\mathrm{d}T}{\mathrm{d}P}\right)_S = \frac{T\sum_i \frac{n_i C_{Pi}\gamma_i}{K_{Si}}}{\sum_i n_i C_{Pi}}, \tag{2.20}$$

where the subscripts correspond to the i th phase, C_P is the heat capacity at constant pressure of a phase, and the other symbols are as defined above. Integrating this ODE requires a choice in anchor temperature (T_0) at the top of the lower mantle (or including this as a parameter in an inversion). As the adiabatic geotherm is dependent on the thermoelastic parameters at high pressures and temperatures, it is dependent on the equation of state used.

2.5.3 Seismic Models

BurnMan allows for direct visual and quantitative comparison with seismic velocity models. Various ways of plotting can be found in the examples. Quantitative misfits between two profiles include an L2-norm and a chi-squared misfit, but user defined norms can be implemented. A seismic model in BurnMan is an object that provides pressure, density, and seismic velocities (V_P, V_Φ, V_S) as a function of depth.

To compare to seismically constrained profiles, BurnMan provides the 1D seismic velocity model PREM [DA81]. One can choose to evaluate $V_P, V_\Phi, V_S, \rho, K_S$ and/or G. The user can input their own seismic profile, an example of which is included using AK135 [KEB95].

Besides standardized 1D radial profiles, one can also compare to regionalized average profiles for the lower mantle. This option accommodates the observation that the lowermost mantle can be clustered into two regions, a 'slow' region, which represents the so-called Large Low Shear Velocity Provinces, and 'fast' region, the continuous surrounding region where slabs might subduct [LCDR12]. This clustering as well as the averaging of the 1D model occurs over five tomographic S wave velocity models (SAW24B16: [MegninR00]; HMSL-S: [HMSL08]; S362ANI: [KED08]; GyPSuM: [SFBG10]; S40RTS: [RDvHW11]). The strongest deviations from PREM occur in the lowermost 1000 km. Using the 'fast' and 'slow' S wave velocity profiles is therefore most important when interpreting the lowermost mantle. Suggestion of compositional variation between these regions comes from seismology [HW12, TRCT05] as well as geochemistry [DCT12, JCK+10]. Based on thermo-chemical convection models, [SDG11] also show that averaging profiles in thermal boundary layers may cause problems for seismic interpretation.

We additionally apply cluster analysis to and provide models for P wave velocity based on two tomographic models (MIT-P08: [LvdH08]; GyPSuM: [SMJM12]). The clustering results correlate well with the fast and slow regions for S wave velocities; this could well be due to the fact that the initial model for the P wave velocity models is scaled from S wave tomographic velocity models. Additionally, the variations in P wave velocities are a lot smaller than for S waves. For this reason using these adapted models is most important when comparing the S wave velocities.

While interpreting lateral variations of seismic velocity in terms of composition and temperature is a major goal [MCD+12, TDRY04], to determine the bulk composition the current challenge appears to be concurrently fitting absolute P and S wave velocities and incorporate the significant uncertainties in mineral physical parameters).

CHAPTER

THREE

THE BURNMAN TUTORIAL

This tutorial introduces users to the main classes and some of the important functions implemented in BurnMan. The tutorial is split into several parts:

The BurnMan Tutorial

3.1 Part 1: Material Classes

This file is part of BurnMan - a thermoelastic and thermodynamic toolkit for the Earth and Planetary Sciences

3.1.1 Introduction

This ipython notebook is the first in a series designed to introduce new users to the code structure and functionalities present in BurnMan.

Demonstrates

1. burnman.Mineral: Equations of state, property modification schemes, initialization, interrogating properties at given pressure and temperature.
2. burnman.CombinedMineral: Initialization (otherwise similar to mineral).
3. burnman.Solution: Formulations (ideal, asymmetric, subregular), initialization, interrogating properties at given pressure, temperature and composition.
4. burnman.Composite: Initialization, interrogating properties at given pressure, temperature, phase proportions and using different seismic averaging schemes.

Everything in BurnMan and in this tutorial is defined in SI units.

3.1.2 Acknowledgments

The authors are grateful to M. Ghiorso for useful discussions. R.M. would like to thank B. Watterson for the concept of Planet Zog (see Part 3: Layers and Planets).

This project was initiated at, and follow-up research support was received through, CIDER (NSF FESD grant 1135452). The development of BurnMan has been supported by the Computational Infrastructure for Geodynamics initiative (CIG), through the Science and Technologies Funding Council (U.K.) under Award No. ST/R001332/1 and through the Natural Environment Research Council (U.K.) Large Grant MC-squared (Award No. NE/T012633/1). The authors have also received support from the University of California - Davis.

3.1.3 Getting started with BurnMan

Our first task is to import BurnMan. If you haven't yet installed the current version, you can do this by typing `pip install -e .` from the top-level directory of the repository. Alternatively, you could just uncomment (remove the leading # from) the first line in the following code block. The second line in the code block imports the BurnMan module.

[1]:
```
#!pip install -q -e ..
import burnman
```

```
Warning: No module named 'cdd'. For full functionality of BurnMan, please␣
↪install pycddlib.
```

3.1.4 Types of BurnMan Material objects

The BurnMan package allows users to define and use objects that represent different kinds of Materials. The most important classes of Material are named Mineral, Solution and Composite. In the following subsections, we show how users can create objects of each type, set their state (pressure and temperature) and composition, and interrogate them for their material properties.

3.1.5 Mineral objects

Mineral objects are the building blocks for more complex objects in BurnMan. These objects are intended to represent minerals (or melts, or fluids) of fixed composition, with a well-defined equation of state that defines the relationship between the current state (pressure and temperature) of the mineral and its thermodynamic potentials and derivatives (such as Gibbs energy, volume and entropy).

Mineral objects are initialized with a "params" dictionary containing all of the parameters required by the desired equation of state and an optional "property modifiers" argument. Here we initialize a generic Mineral, just to show the general structure:

[2]:
```
mineral_object = burnman.Mineral(params={}, property_modifiers=[[]])
```

The required keys in the parameters dictionary depends on the equation of state, described in the following section.

3.1.5.1 Equations of state

BurnMan identifies the desired equation of state by checking the value of the string parameter "equation_of_state" in the parameters dictionary that is passed as an argument to the Mineral class. BurnMan currently contains implementations of the following static equations of state (i.e., equations of state with no temperature dependence): - "bm2", "bm3" and "bm4": Birch-Murnaghan (2nd, 3rd and 4th order) - "mt": Modified Tait - "morse": Morse potential - "vinet": Vinet (originally the Rydberg equation of state) - "rkprime": Reciprocal K-prime

And also the following thermal equations of state: - "mgd2" and "mgd3": Mie-Debye-Grueneisen equation of state (second and third order expansions for the shear modulus) - "slb2" and "slb3": Stixrude and Lithgow-Bertelloni (2011; second and third order expansions for the shear modulus) - "hp_tmt": Thermal Modified Tait (Holland and Powell; 2011) - "dks_s": de Koker and Stixrude (2013; solids) - "dks_l": de Koker and Stixrude (2013; liquids) - "cork": Compensated Redlich-Kwong equation of state - "aa": Anderson and Ahrens (1998) - "brosh_calphad": Brosh et al. (2007)

Each equation of state assumes the presence of a different set of keys in the "params" dictionary. These keys are checked and validated on initialization. Two important parameters for most mineral objects are the chemical formula of the mineral and its molar mass, which can be calculated using the functions "dictionarize_formula" and "formula_mass".

[3]:
```
from burnman.tools.chemistry import dictionarize_formula, formula_mass

forsterite_formula = dictionarize_formula('Mg2SiO4')
molar_mass = formula_mass(forsterite_formula)

print(f'Formula in dictionary form: {forsterite_formula}')
print(f'Molar mass: {molar_mass:.5f} kg')
```

```
Formula in dictionary form: {'Mg': 2.0, 'Si': 1.0, 'O': 4.0}
Molar mass: 0.14069 kg
```

Below, we demonstrate the creation of a forsterite object for the Stixrude and Lithgow-Bertelloni (2011) equation of state which uses a 3rd order expansion for the shear modulus (equation_of_state = 'slb3').

[4]:
```
forsterite_params = {'name': 'Forsterite',
                     'formula': forsterite_formula,
                     'equation_of_state': 'slb3',
                     'F_0': -2055403.0,
                     'V_0': 4.3603e-05,
                     'K_0': 1.279555e+11,
                     'Kprime_0': 4.21796,
                     'Debye_0': 809.1703,
                     'grueneisen_0': 0.99282,
                     'q_0': 2.10672,
                     'G_0': 81.6e9,
                     'Gprime_0': 1.46257,
                     'eta_s_0': 2.29972,
                     'n': sum(forsterite_formula.values()),
```

(continues on next page)

(continued from previous page)

```
                    'molar_mass': molar_mass}

forsterite = burnman.Mineral(params=forsterite_params)
print(forsterite.params)
```

```
{'name': 'Forsterite', 'formula': {'Mg': 2.0, 'Si': 1.0, 'O': 4.0}, 'equation_of_
↪state': 'slb3', 'F_0': -2055403.0, 'V_0': 4.3603e-05, 'K_0': 127955500000.0,
↪'Kprime_0': 4.21796, 'Debye_0': 809.1703, 'grueneisen_0': 0.99282, 'q_0': 2.
↪10672, 'G_0': 81600000000.0, 'Gprime_0': 1.46257, 'eta_s_0': 2.29972, 'n': 7.0,
↪ 'molar_mass': 0.14069310000000002, 'T_0': 300.0, 'E_0': 0.0, 'P_0': 0.0}
```

3.1.5.2 Property modifiers

Thermodynamic models of minerals can include modifications to the properties predicted by the underlying equation of state. Often, this is done to approximate physical processes which were neglected during development of the equation of state.

BurnMan allows users to apply modifications to the Gibbs energy $\mathcal{G}(P,T)$ via a "property_modifiers" attribute. This attribute takes the form of a list of different modifiers, which are themselves each composed of a list consisting of an identifying string and a dictionary containing the required parameters. The following modifiers are currently implemented in BurnMan: - "linear": Linear in pressure and temperature (i.e. $\Delta\mathcal{G} = \Delta\mathcal{E} - T\Delta S + P\Delta V$) - "landau": A Landau order-disorder transition (Putnis, 1992) - "landau_hp": A Landau order-disorder transition (Holland and Powell, 2011) - "bragg_williams": A Bragg-Williams order-disorder transition - "magnetic_chs": A magnetic order-disorder transition

The following code block implements the Landau transition for hematite as given in the Holland et al. (2018) database (ds6.33).

[5]:
```
# Dictionary with parameters for equation of state.
hematite_params = {'name': 'hematite',
                   'formula': {'Fe': 2.0, 'O': 3.0},
                   'equation_of_state': 'hp_tmt',
                   'H_0': -825420.0,
                   'S_0': 87.4,
                   'V_0': 3.027e-05,
                   'Cp': [163.9, 0.0, -2257200.0, -657.6],
                   'a_0': 2.79e-05,
                   'K_0': 223000e6,
                   'Kprime_0': 4.04,
                   'Kdprime_0': -1.8e-11,
                   'n': 5.0,
                   'molar_mass': 0.1596882}

# Dictionary for Landau transition which modifieds the Gibbs energy
hematite_property_modifiers = [['landau_hp', {'P_0': 1.e5,
                                              'T_0': 298.15,
```

(continues on next page)

(continued from previous page)

```
                                                'Tc_0': 955.0,
                                                'S_D': 15.6,
                                                'V_D': 0.0}]]

# Initialise mineral with a property modifier
hematite = burnman.Mineral(params=hematite_params,
                           property_modifiers=hematite_property_modifiers)
```

3.1.5.3 The CombinedMineral class

In petrology, we are often interested in phases for which we have little experimental or theoretical information. One common example is when we want to approximate the properties of an ordered phase relative to its disordered counterparts. In many cases, a reasonable procedure is to make a mechanical mixture of the known phases, such that they are the correct composition of the unknown phase, and then apply a linear correction to the Gibbs energy of the phase (i.e. $\Delta\mathcal{G} = \Delta\mathcal{E} - T\Delta S + P\Delta V$). In BurnMan, we do this using the CombinedMineral class. In the lines below, we create an ordered orthopyroxene with composition $MgFeSi_2O_6$ from a 50:50 mixture of enstatite and ferrosilite. We make this compound 6 kJ/mol more stable than the mechanical mixture.

[6]:
```
from burnman import CombinedMineral
from burnman.minerals import HP_2011_ds62
fe_mg_orthopyroxene = CombinedMineral(name='ordered ferroenstatite',
                                      mineral_list=[HP_2011_ds62.en(),
                                                    HP_2011_ds62.fs()],
                                      molar_amounts=[0.5, 0.5],
                                      free_energy_adjustment=[-6.e3, 0., 0.])
print(fe_mg_orthopyroxene.formula)
```

```
Counter({'O': 6.0, 'Si': 2.0, 'Mg': 1.0, 'Fe': 1.0})
```

3.1.5.4 Implemented datasets

BurnMan already includes several published mineral datasets. These can be found in the "minerals" subdirectory of BurnMan, and include the Stixrude and Lithgow-Bertelloni (2011) dataset, Holland et al. (2013) and Jennings et al. (2015) datasets. The script misc/table_mineral_library.py will list all minerals included for each equation of state.

Mineral objects can be initialised from these datasets using commands like the following:

[7]:
```
from burnman import minerals

fo_SLB = minerals.SLB_2011.forsterite()
fo_HP = minerals.HP_2011_ds62.fo()
```

The "()" at the end of each call indicate that these commands initialize an object from a class. This is done so that users can create multiple distinct objects with the same material parameters but potentially different

states.

3.1.5.5 Interrogating the properties of a Mineral object at a given pressure and temperature

Once an object has been initialised, the user can set its state (pressure in Pa and temperature in K):

[8]:
```
fo_SLB.set_state(1.e9, 1000)
```

The thermodynamic and seismic properties of the object at those conditions can then by returned as attributes of the object (returned in SI units):

[9]:
```
from burnman.tools.chemistry import formula_to_string
print(f'{fo_SLB.name} (SLB2011)')
print(f'Formula: {formula_to_string(fo_SLB.formula)}')
print(f'Gibbs: {fo_SLB.gibbs:.3e} J/mol')
print(f'S: {fo_SLB.molar_entropy:.3e} J/K/mol')
print(f'V_p: {fo_SLB.v_p:.3e} m/s')
```

```
Forsterite (SLB2011)
Formula: Mg2Si04
Gibbs: -2.148e+06 J/mol
S: 2.747e+02 J/K/mol
V_p: 8.308e+03 m/s
```

It is common for users to want to return properties over a range of states. BurnMan makes this easy through the "evaluate" method of the Material class. In the following lines, we investigate the properties of quartz as defined by Stixrude and Lithgow Bertelloni (2011):

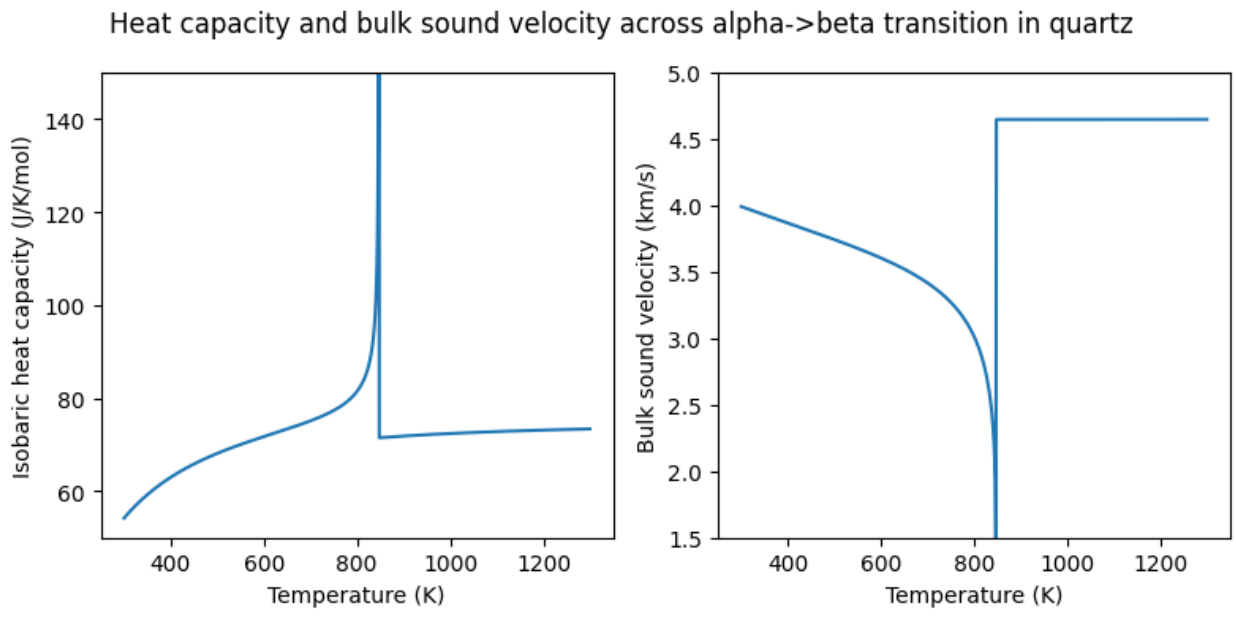
[10]:
```
# Necessary imports for creating the P, T arrays and plotting
import numpy as np
import matplotlib.pyplot as plt

# Temperatures and pressures at which to evaluate properties
temperatures = np.linspace(300., 1300., 1001)
pressures = 1.e5*np.ones_like(temperatures)

# Desired properties
properties = ['molar_heat_capacity_p', 'v_phi']

# Initialize a quartz object and print the parameters dictionary
# and property modifiers
qtz = minerals.SLB_2011.quartz()
print(qtz.params)
print(qtz.property_modifiers)

# Return arrays containing the desired properties
isobaric_heat_capacities, bulk_sound_velocities = qtz.evaluate(properties,␣
```

(continues on next page)

(continued from previous page)

```
↪pressures, temperatures)

# Plot the properties
fig = plt.figure(figsize=(8, 4))
fig.suptitle('Heat capacity and bulk sound velocity across alpha->beta␣
↪transition in quartz')
ax = [fig.add_subplot(1, 2, i) for i in range(1, 3)]
ax[0].plot(temperatures, isobaric_heat_capacities)
ax[1].plot(temperatures, bulk_sound_velocities/1000.)

for i in range(2):
  ax[i].set_xlabel('Temperature (K)')
ax[0].set_ylabel('Isobaric heat capacity (J/K/mol)')
ax[1].set_ylabel('Bulk sound velocity (km/s)')
ax[0].set_ylim(50., 150.)
ax[1].set_ylim(1.5, 5.)
fig.tight_layout()
fig.subplots_adjust(top=0.88)
```

```
{'name': 'Quartz', 'formula': {'Si': 1.0, 'O': 2.0}, 'equation_of_state': 'slb3',
↪ 'F_0': -858853.4, 'V_0': 2.367003e-05, 'K_0': 49547430000.0, 'Kprime_0': 4.
↪33155, 'Debye_0': 816.3307, 'grueneisen_0': -0.00296, 'q_0': 1.0, 'G_0':␣
↪44856170000.0, 'Gprime_0': 0.95315, 'eta_s_0': 2.36469, 'n': 3.0, 'molar_mass':
↪ 0.0600843, 'T_0': 300.0, 'E_0': 0.0, 'P_0': 0.0}
[['landau', {'Tc_0': 847.0, 'S_D': 5.164, 'V_D': 1.222e-06}]]
```



In the figure produced by the code above, one can see the effect of the alpha->beta transition in quartz (also known as the quartz inversion). This is a displacive transition from trigonal to hexagonal symmetry that can be modelled (as done here) by a Landau-type transition.

To give users confidence that BurnMan is outputting accurate derivative properties, we include a tool that checks the thermodynamic self-consistency of the equation of state:

[11]:
```
from burnman.tools.eos import check_eos_consistency
assert(check_eos_consistency(qtz, 1.e9, 1000., verbose=False))
```

3.1.6 Solution objects

The Solution class in BurnMan allows users to define materials that have crystal chemical sites that can be occupied by multiple species. The species occupancies on the sites can be varied by changing the proportions of a number of ‘’independent” endmembers.

Currently, all the models implemented in BurnMan are of “Bragg-Williams” type (Bragg and Williams, 1930s). This means that the thermodynamic properties of solutions are uniquely defined by the *average* occupancies of species on each site (Myhill and Connolly, 2021). These models are therefore unable to accurately account for local interactions that give rise to **short-range order**, but Bragg-Williams models with multiple sites *can* provide a first-order approximation to the energetic effects of **long-range order**. From a microscopic standpoint, the distinction is an artificial one, since long-range order arises from local interactions (e.g. Bethe 1934, 1935). However, a detailed treatment of order-disorder requires either complex models (Kikuchi, 1951) or ab-initio simulations that have their own dedicated software packages (e.g. VASP).

3.1.6.1 Model formalisms

The ideal model

The simplest solution model implemented in BurnMan is the ideal solution model. In this solution, the excess Gibbs energy of mixing is purely configurational in nature. Mathematically:

$\Delta \mathcal{G}^{\text{mix}} = -RT \sum_i m_i \sum_j p_{ij} \ln p_{ij}$

where R is the gas constant (in J/K/mol), T is the temperature (in K), m_i is the multiplicity of site i and p_{ij} is the proportion of species j on site i.

The following code initializes a binary ideal solution model between pyrope and almandine. The solution_model argument defines the solution model itself. The endmembers argument must be a list of lists containing all the endmembers of the solution. The first item in each inner list should contain a Mineral object corresponding to the endmember of interest. The second item should be a chemical formula. The exchange sites in this formula are denoted by square brackets, followed by the multiplicities m_i (optional if $m_i = 1$). Everything not contained within an “[...]m” block is ignored.

[12]:
```
from burnman import Solution, minerals
from burnman.classes.solutionmodel import IdealSolution

ideal_garnet = Solution(name = 'Ideal pyrope-almandine garnet',
                        solution_model = IdealSolution(endmembers = [[minerals.
↪HP_2011_ds62.py(),
```

(continues on next page)

(continued from previous page)

```
↪'[Mg]3[Al]2Si3O12'],
                                                                    [minerals.
↪HP_2011_ds62.alm(),

↪'[Fe]3[Al]2Si3O12']]
                                                          ),
                        molar_fractions = [0.5, 0.5])
```

Each species on each site must begin with a capital letter, but the string does not have to correspond to an element. Partial occupancies are allowed. For example, [Al0.5Fef0.5] would be valid for an endmember with both Al^{3+} and Fe^{3+} on the same site. [Vac] would be valid for a vacancy, whereas [v] would not. The molar_fractions argument in the initialization above is optional.

As of BurnMan 1.1, ideality does not require that the multiplicity of each site is fixed between endmembers. This allows the implementation of Temkin-type models (Temkin, 1945), which are used to model melts. For example, here is the ideal part of the melt model proposed by Holland et al. (2018), restricted to the CMAS system (nonideality and changes in endmember energies have been ignored):

The (a)symmetric model

Another solution model formalism implemented in BurnMan is the asymmetric model (Holland and Powell, 2003). This model is an extension of the regular solution model (Wohl, 1946), with the extension following van Laar (1906).

The general idea is that there are interaction energies W_{ij} associated with each binary in the solution model. Those binary interactions are modified by a set of α_k parameters, such that:

$\Delta \mathcal{G}^{\text{mix}} = \Delta \mathcal{G}^{\text{mix,ideal}} + \Delta \mathcal{G}^{\text{mix,nonideal}}$

and

$\Delta \mathcal{G}^{\text{mix,nonideal}} = (\sum_l \alpha_l p_l)(\sum_i^{n-1} \sum_{j=i+1}^{n} \phi_i \phi_j \frac{2w_{ij}}{\alpha_i + \alpha_j})$

where

$\phi_i = \frac{\alpha_i p_i}{\sum_k \alpha_k p_k}$

$\Delta \mathcal{G}^{\text{mix,ideal}}$ is caalculated using the same expressions as for the ideal model. The following code block initialises an asymmetric garnet model in the system CFMASO (taken from the Holland and Powell, 2011 dataset; ds6.2).

[13]:
```
from burnman.classes.solutionmodel import AsymmetricRegularSolution

g2 = Solution(name='asymmetric garnet (ThermoCalc ds6.2)',
              solution_model=AsymmetricRegularSolution(endmembers=[[minerals.HP_
↪2011_ds62.py(), '[Mg]3[Al]2Si3O12'],
                                                                   [minerals.HP_
↪2011_ds62.alm(), '[Fe]3[Al]2Si3O12'],
                                                                   [minerals.HP_
```

(continues on next page)

(continued from previous page)

```
↪2011_ds62.gr(), '[Ca]3[Al]2Si3O12'],
                                                                    [minerals.HP_
↪2011_ds62.andr(), '[Ca]3[Fe]2Si3O12']],
                                                          alphas = [1.0, 1.0, 2.7,␣
↪2.7],
                                                          energy_interaction = [[2.
↪5e3, 31.e3, 53.2e3],
                                                                                [5.
↪e3, 37.24e3],
                                                                                [2.
↪e3]]))
```

In the case that all α_i are equal to each other, the asymmetric model becomes a symmetric model. BurnMan allows users to specify this type of model by setting solution_model=SymmetricRegularSolution(). In this case, the alphas argument does not need to be passed to SymmetricRegularSolution().

The subregular model

BurnMan also contains an implementation of the subregular model (Helffrich and Wood, 1989). In this model, the excess nonideal Gibbs energy is expressed as a cubic polynomial in endmember proportions:

$G^* = p_i G_i^{\text{mbr}} + p_i p_j W_{ij}^{\text{binary}}(1 + p_j - p_i)/2 + p_i p_j p_k W_{ijk}^{\text{ternary}}$

where the binary terms must be equal to zero when $i = j$ and the ternary terms can only be non-zero for $i < j < k$.

[14]:
```
from burnman.classes.solutionmodel import SubregularSolution

# Parameters from Ganguly et al. (1996), converted to SI units
Wh_1bar = [[[2117., 695.], [9834., 21627.], [12083., 12083.]],
           [[6773., 873.], [539., 539.]],
           [[0., 0.]]]
Wv = [[[0.07e-5, 0.], [0.058e-5, 0.012e-5], [0.04e-5, 0.03e-5]],
      [[0.03e-5, 0.], [0.04e-5, 0.01e-5]],
      [[0., 0.]]]
Ws = [[[0., 0.], [5.78, 5.78], [7.67, 7.67]],
      [[1.69, 1.69], [0., 0.]],
      [[0., 0.]]]

# We now convert from 1 bar enthalpy, entropy and volume (1 cation basis)
# to energy, entropy and volume (3 cation basis)
mult = lambda x, n: [[[v*n for v in i] for i in j] for j in x]
add = lambda x, y: [[[x[i][j][k] + y[i][j][k] for k in range(len(x[i][j]))]
                     for j in range(len(x[i]))] for i in range(len(x))]

energy_interaction = add(mult(Wh_1bar, 3.), mult(Wv, -3.e5))
```

(continues on next page)

(continued from previous page)

```
volume_interaction = mult(Wv, 3.)
entropy_interaction = mult(Ws, 3.)

g3 = Solution(name='CFMnMAS garnet (Ganguly et al., 1996)',
              solution_model=SubregularSolution(
              endmembers=[[minerals.HP_2011_ds62.py(), '[Mg]3[Al]2Si3O12'],
                          [minerals.HP_2011_ds62.alm(), '[Fe]3[Al]2Si3O12'],
                          [minerals.HP_2011_ds62.gr(), '[Ca]3[Al]2Si3O12'],
                          [minerals.HP_2011_ds62.spss(), '[Mn]3[Al]2Si3O12']],
              energy_interaction=energy_interaction,
              volume_interaction=volume_interaction,
              entropy_interaction=entropy_interaction,
              energy_ternary_terms = [[0, 1, 2, 0.e3]],
              volume_ternary_terms = [[0, 1, 2, 0.e-6]],
              entropy_ternary_terms = [[0, 1, 2, 0.]]))
```

For the model above, the ternary terms are all equal to zero, but in general, the structure of the ternary_terms input is a list of lists, where the inner list corresponds to the three endmember indices $i < j < k$ followed by the value of the interaction term.

3.1.6.2 Implemented datasets

As for the endmembers, BurnMan already contains solution models from multiple datasets. One of these is the Stixrude and Lithgow-Bertelloni (2011) dataset. In the code-block below, we initialize an object using the three-endmember (FMAS) bridgmanite solution from that publication.

[15]:
```
bdg = minerals.SLB_2011.mg_fe_bridgmanite()
bdg.endmembers
```

[15]:
```
[[<burnman.minerals.SLB_2011.mg_perovskite at 0x7efd464d9c10>, '[Mg][Si]O3'],
 [<burnman.minerals.SLB_2011.fe_perovskite at 0x7efd46283050>, '[Fe][Si]O3'],
 [<burnman.minerals.SLB_2011.al_perovskite at 0x7efd46283690>, '[Al][Al]O3']]
```

3.1.6.3 Interrogating the properties of a Solution object at a given composition, pressure and temperature

Unlike Minerals, Solutions can vary in composition. Before interrogating properties, it is therefore necessary to set both the state (using the method "set_state") and the composition (using the method "set_composition").

[16]:
```
bdg.set_composition([0.8, 0.1, 0.1])
bdg.set_state(30.e9, 2000.)

print(f'Bridgmanite composition: {dict(bdg.formula)}')
print(f'Bridgmanite volume: {bdg.V:.3e} m')
```

(continues on next page)

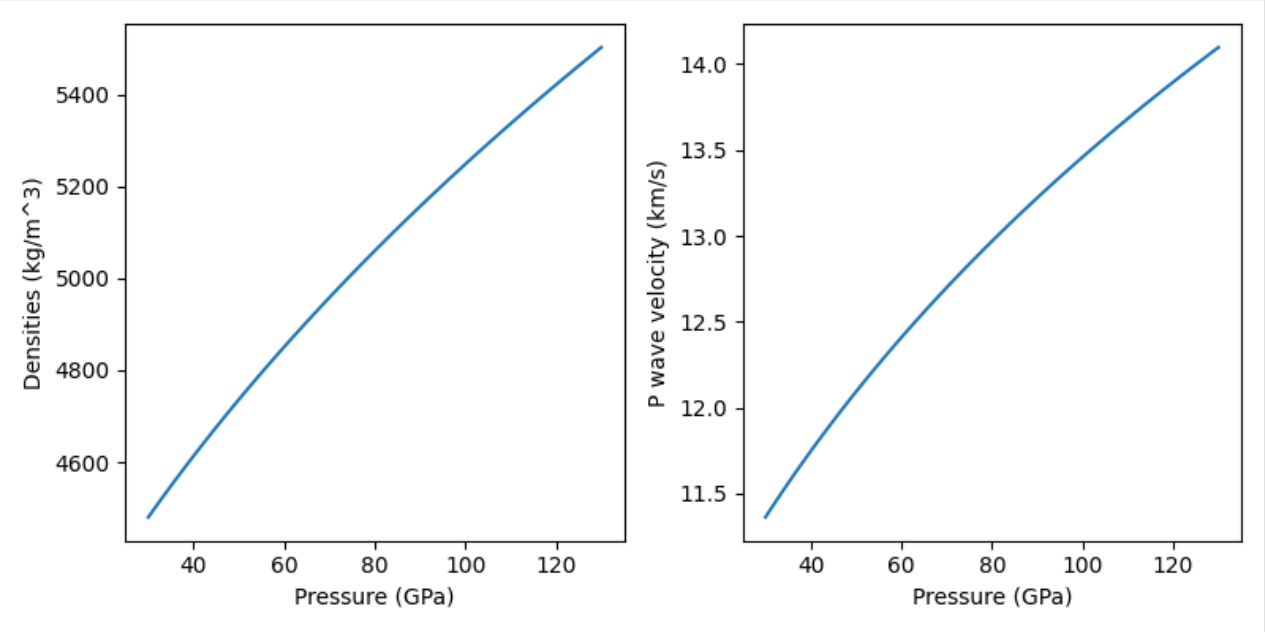
(continued from previous page)

```
print(f'Bridgmanite endmember partial Gibbs energies:')
for pG in bdg.partial_gibbs:
    print(f'   {pG/1e3:.3e} kJ/mol')
```

```
Bridgmanite composition: {'Mg': 0.8, 'Si': 0.9, 'O': 3.0, 'Fe': 0.1, 'Al': 0.2}
Bridgmanite volume: 2.315e-05 m
Bridgmanite endmember partial Gibbs energies:
   -9.690e+02 kJ/mol
   -6.868e+02 kJ/mol
   -1.143e+03 kJ/mol
```

The evaluate method can be used to output properties of solutions at constant composition:

[17]:
```
pressures = np.linspace(30.e9, 130.e9, 101)
temperatures = 2000.*np.ones_like(pressures)
densities, p_wave_velocities = bdg.evaluate(['rho', 'v_p'], pressures,␣
↪temperatures)

# The following lines do the plotting
fig = plt.figure(figsize=(8, 4))
ax = [fig.add_subplot(1, 2, i) for i in range(1, 3)]

ax[0].plot(pressures/1.e9, densities)
ax[1].plot(pressures/1.e9, p_wave_velocities/1000.)

for i in range(2):
  ax[i].set_xlabel('Pressure (GPa)')
ax[0].set_ylabel('Densities (kg/m^3)')
ax[1].set_ylabel('P wave velocity (km/s)')
fig.tight_layout()
```



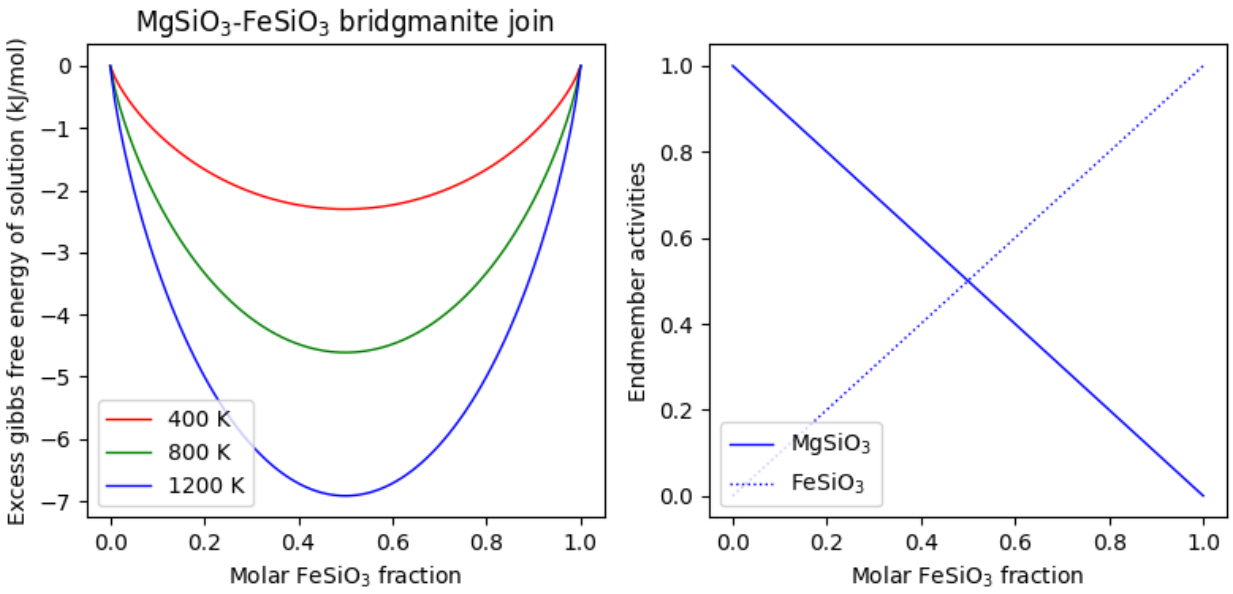
The "set_composition" method must be used every time the Solution composition is to be changed:

[18]:
```
import numpy as np
import matplotlib.pyplot as plt

comp = np.linspace(1e-5, 1.-1e-5, 101)

bdg_excess_gibbs_400 = np.empty_like(comp)
bdg_excess_gibbs_800 = np.empty_like(comp)
bdg_excess_gibbs_1200 = np.empty_like(comp)

bdg_activities_400 = np.empty((101, 3))
bdg_activities_800 = np.empty((101, 3))
bdg_activities_1200 = np.empty((101, 3))

pressure = 1.e9
for i, c in enumerate(comp):
    molar_fractions = [1.0 - c, c, 0.]
    bdg.set_composition(molar_fractions)
    bdg.set_state(pressure, 400.)
    bdg_excess_gibbs_400[i] = bdg.excess_gibbs
    bdg_activities_400[i] = bdg.activities
    bdg.set_state(pressure, 800.)
    bdg_excess_gibbs_800[i] = bdg.excess_gibbs
    bdg_activities_800[i] = bdg.activities
    bdg.set_state(pressure, 1200.)
    bdg_excess_gibbs_1200[i] = bdg.excess_gibbs
    bdg_activities_1200[i] = bdg.activities
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(8, 4))
ax = [fig.add_subplot(1, 2, i) for i in range(1, 3)]
ax[0].plot(comp, bdg_excess_gibbs_400/1000., 'r-', linewidth=1., label='400 K')
ax[0].plot(comp, bdg_excess_gibbs_800/1000., 'g-', linewidth=1., label='800 K')
ax[0].plot(comp, bdg_excess_gibbs_1200/1000., 'b-', linewidth=1., label='1200 K')

ax[1].plot(comp, bdg_activities_1200[:,0], 'b-', linewidth=1., label='MgSiO$_3$')
ax[1].plot(comp, bdg_activities_1200[:,1], 'b:', linewidth=1., label='FeSiO$_3$')


ax[0].set_title("MgSiO$_3$-FeSiO$_3$ bridgmanite join")
ax[0].set_ylabel("Excess gibbs free energy of solution (kJ/mol)")
ax[1].set_ylabel("Endmember activities")

for i in range(2):
  ax[i].set_xlabel("Molar FeSiO$_3$ fraction")
  ax[i].legend(loc='lower left')

fig.tight_layout()
plt.show()
```



3.1.7 Composite objects

The third major class derived from Material is the Composite class. This class is designed to represent collections of other Materials, which are typically objects of type Mineral and Solution, but can be any Material, including other Composites. A list of Materials is passed to the argument "phases" on initialization of a Composite object.

The properties of Composite materials can be interrogated in a similar manner to Solutions. To set the molar fractions of the material, either pass an array of fractions as the argument "fractions" on initialization, or use the "set_fractions" method at any time after initialization. As before, "set_state" is used to set the pressure and temperature of the object.

[19]:
```
from burnman import Composite

bdg = minerals.SLB_2011.mg_fe_bridgmanite()
fper = minerals.SLB_2011.ferropericlase()
bdg.set_composition([0.9, 0.1, 0.0])
fper.set_composition([0.8, 0.2])

assemblage = Composite(phases=[bdg, fper],
                       fractions=[0.5, 0.5],
                       fraction_type='molar',
                       name='rock')

pressure = 30.e9
temperature = 2000.
assemblage.set_fractions([0.5, 0.5])
assemblage.set_state(pressure, temperature)

print(f'Assemblage density at {pressure/1.e9} GPa and {temperature} K:
↪{assemblage.density:.1f} kg/m^3')
```

```
Assemblage density at 30.0 GPa and 2000.0 K: 4491.0 kg/m^3
```

As for the Solution class, the properties of Composites at fixed phase fraction and phase compositions can be obtined using the evaluate method of the class. In the following code block, we evaluate the seismic properties of our 50:50 molar mixture of bridgmanite and ferropericlase from 30 to 130 GPa at 2000 K.

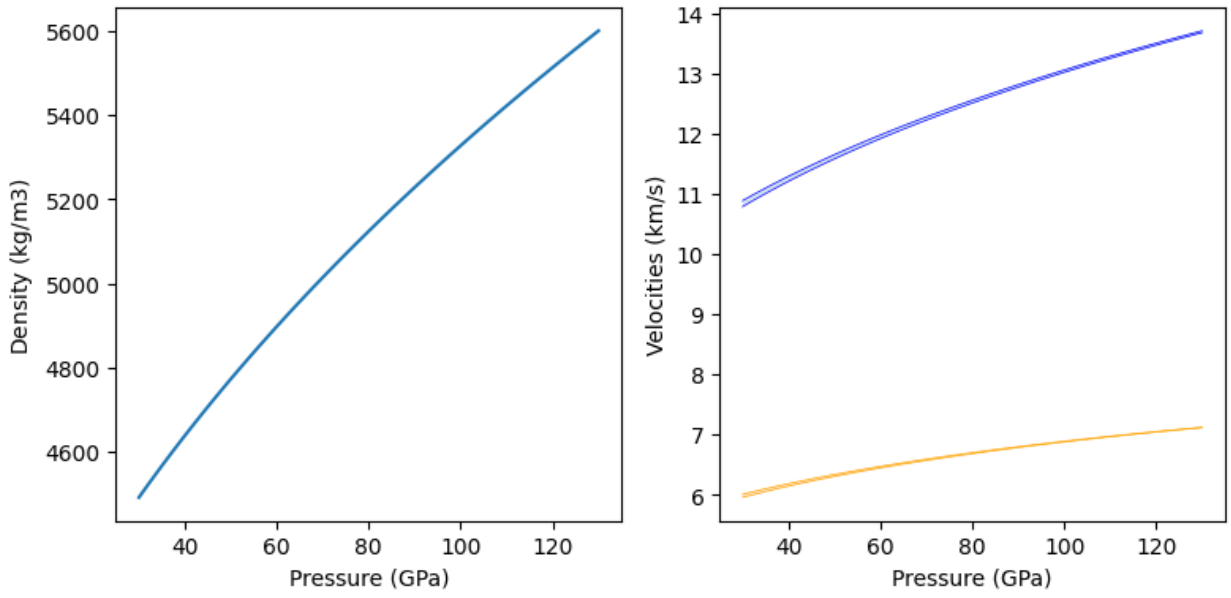
In this code block, we also demonstrate the ability for BurnMan to switch between different schemes for averaging seismic properties. Available schemes are "Voigt", "Reuss", "VoigtReussHill" (the standard scheme, which averages the Voigt and Reuss averages), "HashinShtrikmanLower" and "HashinShtrikmanUpper". The figure shows the difference between the Reuss (lower) and Voigt (upper) bounds on the P-wave and S-wave velocities.

[20]:
```
pressures = np.linspace(30.e9, 130.e9, 101)
temperatures = np.ones(101)*2000.

assemblage.set_averaging_scheme('VoigtReussHill')
densities, Vp_VRH, Vs_VRH = assemblage.evaluate(['rho', 'v_p', 'v_s'],
```

(continues on next page)

(continued from previous page)

```
                                                pressures, temperatures)

assemblage.set_averaging_scheme('Reuss')
Vp_R, Vs_R = assemblage.evaluate(['v_p', 'v_s'], pressures, temperatures)

assemblage.set_averaging_scheme('Voigt')
Vp_V, Vs_V = assemblage.evaluate(['v_p', 'v_s'], pressures, temperatures)


fig = plt.figure(figsize=(8, 4))
ax = [fig.add_subplot(1, 2, i) for i in range(1, 3)]
ax[0].plot(pressures/1.e9, densities)
ax[1].fill_between(pressures/1.e9, Vp_R/1.e3, Vp_V/1.e3, alpha=0.2)
ax[1].fill_between(pressures/1.e9, Vs_R/1.e3, Vs_V/1.e3, alpha=0.2)
ax[1].plot(pressures/1.e9, Vs_R/1.e3, color='orange', linewidth=0.5)
ax[1].plot(pressures/1.e9, Vs_V/1.e3, color='orange', linewidth=0.5, label='$V_S$
↪')
ax[1].plot(pressures/1.e9, Vp_R/1.e3, color='blue', linewidth=0.5)
ax[1].plot(pressures/1.e9, Vp_V/1.e3, color='blue', linewidth=0.5, label='$V_P$')

for i in range(2):
  ax[i].set_xlabel('Pressure (GPa)')

ax[0].set_ylabel('Density (kg/m$3$)')
ax[1].set_ylabel('Velocities (km/s)')

fig.tight_layout()
plt.show()
```



In the next part of the tutorial, we will look at BurnMan's Composition class.

The BurnMan Tutorial

3.2 Part 2: The Composition Class

This file is part of BurnMan - a thermoelastic and thermodynamic toolkit for the Earth and Planetary Sciences

3.2.1 Introduction

This ipython notebook is the second in a series designed to introduce new users to the code structure and functionalities present in BurnMan.

Demonstrates

1. burnman.Composition: Defining Composition objects, converting between molar, weight and atomic amounts, changing component bases. and modifying compositions.

Everything in BurnMan and in this tutorial is defined in SI units.

3.2.2 The Composition class

It is quite common in petrology to want to perform simple manipulations on chemical compositions. These manipulations might include: - converting between molar and weight percent of oxides or elements - changing from one compositional basis to another (e.g. 'FeO' and 'Fe2O3' to 'Fe' and 'O') - adding new chemical components to an existing composition in specific proportions with existing components.

These operations are easy to perform in Excel (for example), but errors are surprisingly common, and are even present in published literature. BurnMan's Composition class is designed to make some of these common tasks easy and hopefully less error prone. Composition objects are initialised with a dictionary of component amounts (in any format), followed by a string that indicates whether that composition is given in "molar" amounts or "weight" (more technically mass, but weight is a more commonly used word in chemistry).

[1]:
```
from burnman import Composition

olivine_composition = Composition({'MgO': 1.8,
                                   'FeO': 0.2,
                                   'SiO2': 1.}, 'molar')
```

```
Warning: No module named 'cdd'. For full functionality of BurnMan, please␣
↪install pycddlib.
```

After initialization, the "print" method can be used to directly print molar, weight or atomic amounts. Optional variables control the print precision and normalization of amounts.

[2]:
```
olivine_composition.print('molar', significant_figures=4,
                          normalization_component='SiO2', normalization_amount=1.
↪)
olivine_composition.print('weight', significant_figures=4,
                          normalization_component='total', normalization_
↪amount=1.)
olivine_composition.print('atomic', significant_figures=4,
                          normalization_component='total', normalization_
↪amount=7.)
```

```
Molar composition
FeO: 0.2000
MgO: 1.8000
SiO2: 1.0000
Weight composition
FeO: 0.0977
MgO: 0.4935
SiO2: 0.4087
Atomic composition
Fe: 0.2000
Mg: 1.8000
O: 4.0000
Si: 1.0000
```

Let's do something a little more complicated. When we're making a starting mix for petrological experiments, we often have to add additional components. For example, we add iron as Fe2O3 even if we want a reduced oxide starting mix, because FeO is not a stable stoichiometric compound.

Here we show how to use BurnMan to create such mixes. In this case, let's say we want to create a KLB-1 starting mix (Takahashi, 1986). We know the weight proportions of the various oxides (including only components in the NCFMAS system):

[3]:
```
KLB1 = Composition({'SiO2': 44.48,
                    'Al2O3': 3.59,
                    'FeO': 8.10,
                    'MgO': 39.22,
                    'CaO': 3.44,
                    'Na2O': 0.30}, 'weight')
```

However, this composition is not the composition we wish to make in the lab. We need to make the following changes: - CaO and Na_2O should be added as $CaCO_3$ and Na_2CO_3. - FeO should be added as Fe_2O_3

First, we change the bulk composition to satisfy these requirements. The molar amounts of the existing components are stored in a dictionary "molar_composition", and can be used to determine the amounts of CO2 and O to add to the bulk composition:

[4]:
```
CO2_molar = KLB1.molar_composition['CaO'] + KLB1.molar_composition['Na2O']
O_molar = KLB1.molar_composition['FeO']*0.5
```

(continues on next page)

(continued from previous page)

```
KLB1.add_components(composition_dictionary = {'CO2': CO2_molar,
                                              'O': O_molar},
                    unit_type = 'molar')
```

Then we can change the component set to the oxidised, carbonated compounds and print the desired starting compositions, for 2 g total mass:

[5]:
```
KLB1.change_component_set(['Na2CO3', 'CaCO3', 'Fe2O3', 'MgO', 'Al2O3', 'SiO2'])
KLB1.print('weight', significant_figures=4, normalization_amount=2.)
```

```
Weight composition
Al2O3: 0.0697
CaCO3: 0.1193
Fe2O3: 0.1749
MgO: 0.7620
Na2CO3: 0.0100
SiO2: 0.8642
```

And that's it! The next tutorial will be on making Layer and Planet objects for planetary science applications.

The BurnMan Tutorial

3.3 Part 3: Layers and Planets

This file is part of BurnMan - a thermoelastic and thermodynamic toolkit for the Earth and Planetary Sciences

Copyright (C) 2012 - 2021 by the BurnMan team, released under the GNU GPL v2 or later.

3.3.1 Introduction

This ipython notebook is the third in a series designed to introduce new users to the code structure and functionalities present in BurnMan.

Demonstrates

1. burnman.Layer
2. burnman.Planet

Everything in BurnMan and in this tutorial is defined in SI units.

3.3.2 Building a planet

Planets are, to a good first approximation, layered like an onion. They typically have a core, a mantle and a crust, stratified according to density. Because temperature and pressure are both strongly correlated with depth, major phase transitions are (again, to first order) reasonably treated as being continuous and horizontal.

On Earth, these approximations have led to 1D models of properties, such as PREM (Dziewonski and Anderson, 1981) and AK135 (Kennett, Engdahl and Buland, 1995). These models can be used as a starting point for studies investigating the possible composition and temperature of Earth's deep interior.

On other planets, we do not yet have data anywhere near as good as for Earth, and so interior structure is less well-known. What we do have is gravity at the surface, mass of the planet, and moment of inertia. So the question is then, what might those things tell us about interior mineralogy?

BurnMan includes Layer and Planet classes to help investigate these questions.

3.3.3 The Layer class

The Layer class in BurnMan is designed to represent a spherical shell of a planet. That shell is made of a BurnMan Material object. The user can specify how the pressure and temperature evolve within the Layer.

The following code block creates a lower_mantle layer of mg_bridgmanite and periclase in 80:20 molar proportions, and assigns an adiabatic profile with a temperature of 1900 K at the top of the layer. The pressure is set to be self-consistent; i.e. the gravity and pressure are adjusted to ensure that pressure is hydrostatic.

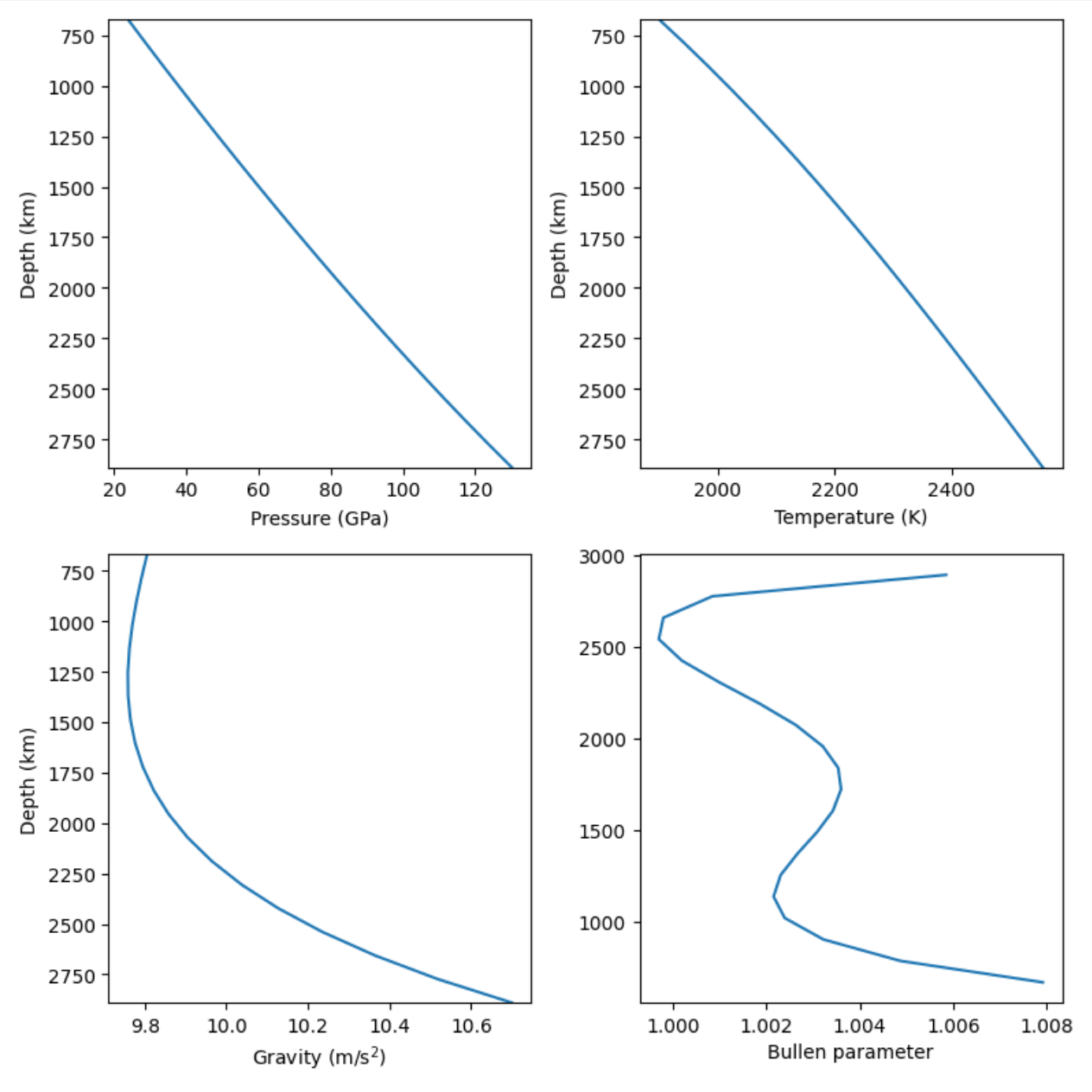
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import burnman
from burnman import Mineral, PerplexMaterial, Composite, Layer, Planet
from burnman import minerals


depths = np.linspace(2890e3, 670e3, 20)
rock = Composite([minerals.SLB_2011.mg_bridgmanite(),
                  minerals.SLB_2011.periclase()],
                 [0.8, 0.2])

lower_mantle = Layer(name='Lower Mantle', radii=6371.e3-depths)
lower_mantle.set_material(rock)
lower_mantle.set_temperature_mode(temperature_mode='adiabatic',
                                  temperature_top=1900.)
lower_mantle.set_pressure_mode(pressure_mode='self-consistent',
                               pressure_top=23.8e9,
                               gravity_bottom=10.7)

# The "make" method does the calculations to make the pressure and gravity self-
↪consistent.
lower_mantle.make()
```

(continues on next page)

(continued from previous page)

```
fig = plt.figure(figsize=(8, 8))
ax = [fig.add_subplot(2, 2, i) for i in range(1, 5)]
ax[0].plot(lower_mantle.pressure/1.e9, 6371.-lower_mantle.radii/1.e3)
ax[1].plot(lower_mantle.temperature, 6371.-lower_mantle.radii/1.e3)
ax[2].plot(lower_mantle.gravity, 6371.-lower_mantle.radii/1.e3)
ax[3].plot(lower_mantle.bullen, 6371.-lower_mantle.radii/1.e3)
for i in range(3):
    ax[i].set_ylim(6371.-lower_mantle.radii[0]/1.e3,
                   6371.-lower_mantle.radii[-1]/1.e3)
    ax[i].set_ylabel('Depth (km)')

ax[0].set_xlabel('Pressure (GPa)')
ax[1].set_xlabel('Temperature (K)')
ax[2].set_xlabel('Gravity (m/s$^2$)')
ax[3].set_xlabel('Bullen parameter')

fig.set_tight_layout(True)
```

```
Warning: No module named 'cdd'. For full functionality of BurnMan, please␣
↪install pycddlib.
```



3.3.4 The Planet class

In a 1D Planet, the pressure, gravity, temperature and temperature gradient at the interfaces between layers must be continuous. In BurnMan, it is possible to collect layers together into a Planet, and have the "make" method of Planet work out how to ensure continuity (at least for pressure, gravity and temperature; for the sake of flexibility, temperature gradient is allowed to be discontinuous).

In the following example, we build Planet Zog, a planet similar to Earth but a little simpler in mineralogical makeup. First, we create an adiabatic inner core. The inner core probably isn't adiabatic, but this is largely unimportant for the innermost layer.

[2]:
```
from burnman import Composition
from burnman.tools.chemistry import formula_mass

# Compositions from midpoints of Hirose et al. (2021), ignoring carbon and␣
↪hydrogen
inner_core_composition = Composition({'Fe': 94.4, 'Ni': 5., 'Si': 0.55, 'O': 0.
↪05}, 'weight')
outer_core_composition = Composition({'Fe': 90., 'Ni': 5., 'Si': 2., 'O': 3.},
↪'weight')


for c in [inner_core_composition, outer_core_composition]:
    c.renormalize('atomic', 'total', 1.)

inner_core_elemental_composition = dict(inner_core_composition.atomic_
↪composition)
outer_core_elemental_composition = dict(outer_core_composition.atomic_
↪composition)
inner_core_molar_mass = formula_mass(inner_core_elemental_composition)
outer_core_molar_mass = formula_mass(outer_core_elemental_composition)
```

[3]:
```
icb_radius = 1220.e3
inner_core = Layer('inner core', radii=np.linspace(0., icb_radius, 21))

hcp_iron = minerals.SE_2015.hcp_iron()
params = hcp_iron.params

params['name'] = 'modified solid iron'
params['formula'] = inner_core_elemental_composition
params['molar_mass'] = inner_core_molar_mass
delta_V = 2.0e-7

inner_core_material = Mineral(params=params,
                              property_modifiers=[['linear',
                                                   {'delta_E': 0.,
                                                    'delta_S': 0.,
                                                    'delta_V': delta_V}]])

# check that the new inner core material does what we expect:
hcp_iron.set_state(200.e9, 4000.)
inner_core_material.set_state(200.e9, 4000.)
assert np.abs(delta_V - (inner_core_material.V - hcp_iron.V)) < 1.e-12

inner_core.set_material(inner_core_material)

inner_core.set_temperature_mode('adiabatic')
```

Now, we create an adiabatic outer core.

[4]:
```
cmb_radius = 3480.e3
outer_core = Layer('outer core', radii=np.linspace(icb_radius, cmb_radius, 21))

liq_iron = minerals.SE_2015.liquid_iron()
params = liq_iron.params

params['name'] = 'modified liquid iron'
params['formula'] = outer_core_elemental_composition
params['molar_mass'] = outer_core_molar_mass
delta_V = -2.3e-7
outer_core_material = Mineral(params=params,
                              property_modifiers=[['linear',
                                                   {'delta_E': 0.,
                                                    'delta_S': 0.,
                                                    'delta_V': delta_V}]])

# check that the new inner core material does what we expect:
liq_iron.set_state(200.e9, 4000.)
outer_core_material.set_state(200.e9, 4000.)
assert np.abs(delta_V - (outer_core_material.V - liq_iron.V)) < 1.e-12

outer_core.set_material(outer_core_material)

outer_core.set_temperature_mode('adiabatic')
```

Now, we assume that there is a single mantle layer that is convecting. We import a PerpleX input table that contains the material properties of pyrolite for this layer. We'll use this for the convecting mantle layer. We apply a perturbed adiabatic temperature profile.

[5]:
```
from burnman import BoundaryLayerPerturbation

lab_radius = 6171.e3 # 200 km thick lithosphere
lab_temperature = 1550.

convecting_mantle_radii = np.linspace(cmb_radius, lab_radius, 101)
convecting_mantle = Layer('convecting mantle', radii=convecting_mantle_radii)

# Import a low resolution PerpleX data table.
fname = '../tutorial/data/pyrolite_perplex_table_lo_res.dat'
pyrolite = PerplexMaterial(fname, name='pyrolite')
convecting_mantle.set_material(pyrolite)

# Here we add a thermal boundary layer perturbation, assuming that the
# lower mantle has a Rayleigh number of 1.e7, and that the basal thermal
# boundary layer has a temperature jump of 840 K and the top
```

(continues on next page)

(continued from previous page)

```
# boundary layer has a temperature jump of 60 K.
tbl_perturbation = BoundaryLayerPerturbation(radius_bottom=cmb_radius,
                                             radius_top=lab_radius,
                                             rayleigh_number=1.e7,
                                             temperature_change=900.,
                                             boundary_layer_ratio=60./900.)

# Onto this perturbation, we add a linear superadiabaticity term according
# to Anderson (he settled on 200 K over the lower mantle)
dT_superadiabatic = 300.*(convecting_mantle_radii - convecting_mantle_radii[-1])/
↪(convecting_mantle_radii[0] - convecting_mantle_radii[-1])

convecting_mantle_tbl = (tbl_perturbation.temperature(convecting_mantle_radii)
                         + dT_superadiabatic)

convecting_mantle.set_temperature_mode('perturbed-adiabatic',
                                       temperatures=convecting_mantle_tbl)
```

And the lithosphere has a user-defined conductive gradient.

[6]:
```
moho_radius = 6341.e3
moho_temperature = 620.

dunite = minerals.SLB_2011.mg_fe_olivine(molar_fractions=[0.92, 0.08])
lithospheric_mantle = Layer('lithospheric mantle',
                            radii=np.linspace(lab_radius, moho_radius, 31))
lithospheric_mantle.set_material(dunite)
lithospheric_mantle.set_temperature_mode('user-defined',
                                         np.linspace(lab_temperature,
                                                     moho_temperature, 31))
```

Finally, we assume the crust has the density of andesine ~ 40% anorthite

[7]:
```
planet_radius = 6371.e3
surface_temperature = 300.
andesine = minerals.SLB_2011.plagioclase(molar_fractions=[0.4, 0.6])
crust = Layer('crust', radii=np.linspace(moho_radius, planet_radius, 11))
crust.set_material(andesine)
crust.set_temperature_mode('user-defined',
                           np.linspace(moho_temperature,
                                       surface_temperature, 11))
```

Everything is ready! Let's make our planet from its consistuent layers.

[8]:
```
planet_zog = Planet('Planet Zog',
                    [inner_core, outer_core,
```

(continues on next page)

(continued from previous page)

```
                    convecting_mantle, lithospheric_mantle,
                    crust], verbose=True)
planet_zog.make()
```

```
Iteration 1 maximum relative pressure error: 9.5e-01
Iteration 2 maximum relative pressure error: 4.8e-01
Iteration 3 maximum relative pressure error: 2.1e-01
Iteration 4 maximum relative pressure error: 8.5e-02
Iteration 5 maximum relative pressure error: 3.4e-02
Iteration 6 maximum relative pressure error: 1.3e-02
Iteration 7 maximum relative pressure error: 5.2e-03
Iteration 8 maximum relative pressure error: 2.0e-03
Iteration 9 maximum relative pressure error: 7.9e-04
Iteration 10 maximum relative pressure error: 3.1e-04
Iteration 11 maximum relative pressure error: 1.2e-04
Iteration 12 maximum relative pressure error: 4.7e-05
Iteration 13 maximum relative pressure error: 1.8e-05
Iteration 14 maximum relative pressure error: 7.1e-06
```

Now we output the mass of the planet and moment of inertia and the mass of the individual layers:

[9]:
```
earth_mass = 5.972e24
earth_moment_of_inertia_factor = 0.3307

print(f'mass = {planet_zog.mass:.3e} (Earth = {earth_mass:.3e})')
print(f'moment of inertia factor= {planet_zog.moment_of_inertia_factor:.4f} '
      f'(Earth = {earth_moment_of_inertia_factor:.4f})')

print('Layer mass fractions:')
for layer in planet_zog.layers:
    print(f'{layer.name}: {layer.mass / planet_zog.mass:.3f}')
```

```
mass = 5.962e+24 (Earth = 5.972e+24)
moment of inertia factor= 0.3307 (Earth = 0.3307)
Layer mass fractions:
inner core: 0.016
outer core: 0.309
convecting mantle: 0.621
lithospheric mantle: 0.047
crust: 0.007
```

BurnMan contains some utility functions that output data in a format readable by seismic codes such as Axisem and Mineos.

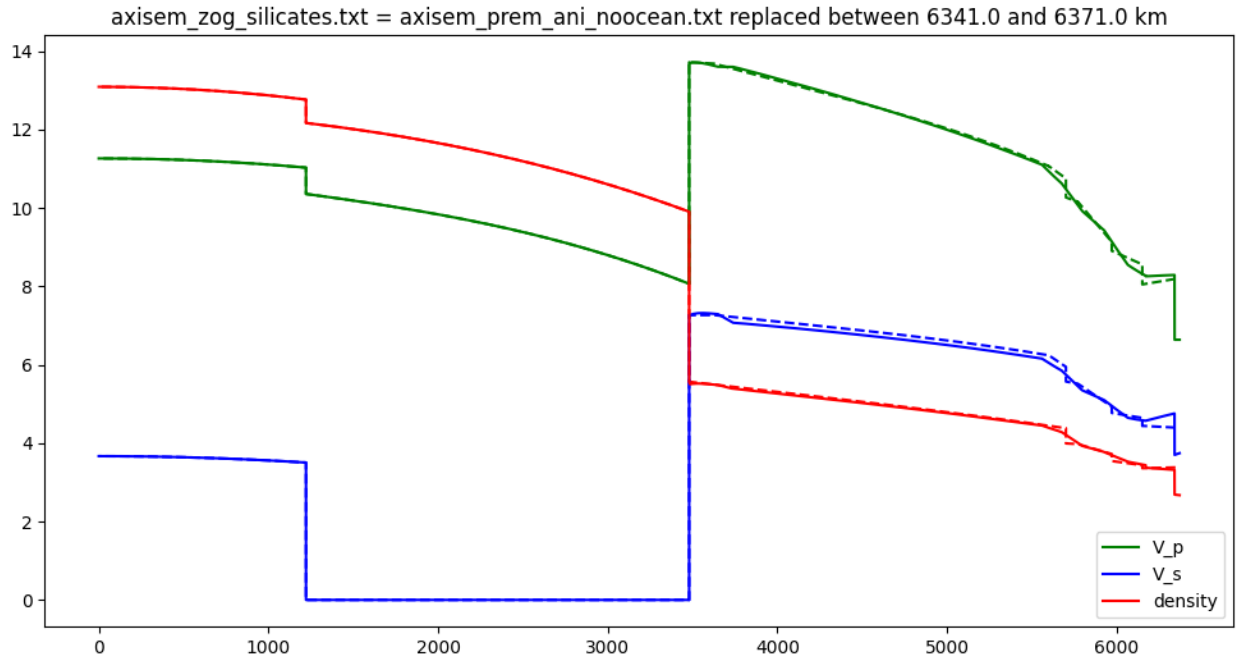
[10]:
```
from burnman.tools.output_seismo import write_axisem_input
from burnman.tools.output_seismo import write_mineos_input
```

(continues on next page)

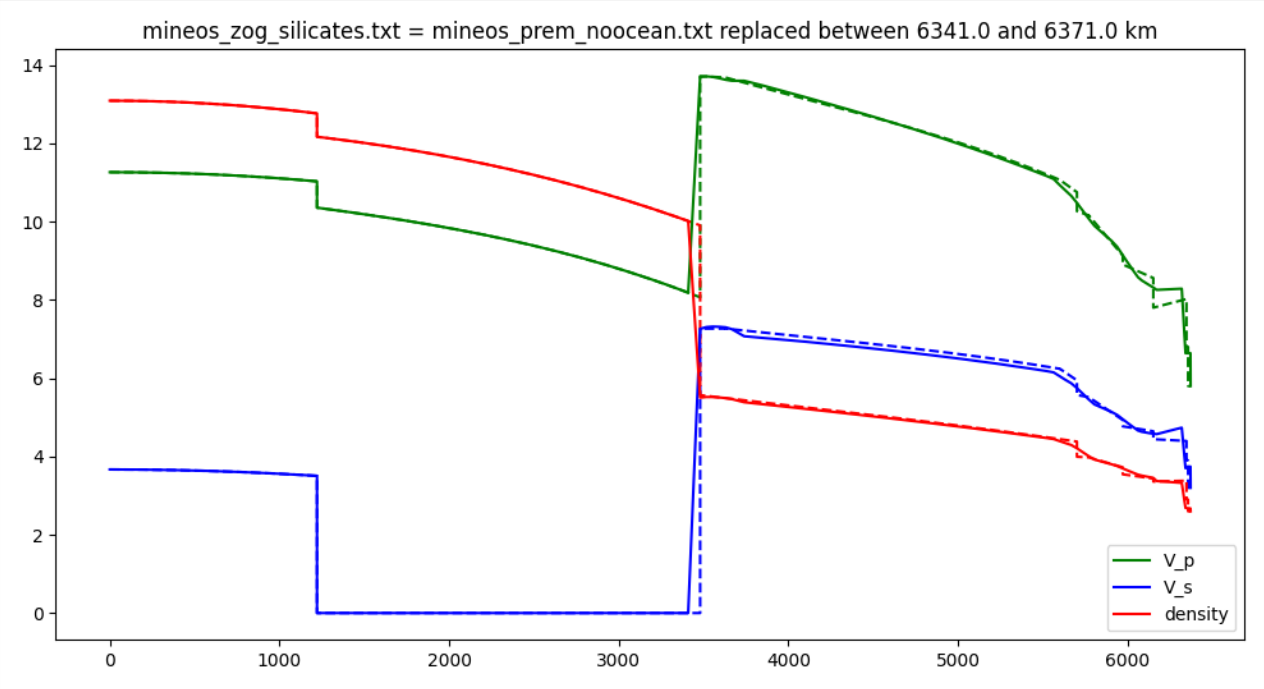
(continued from previous page)

```
write_axisem_input([convecting_mantle, lithospheric_mantle, crust], modelname=
↪'zog_silicates', plotting=True)
write_mineos_input([convecting_mantle, lithospheric_mantle, crust], modelname=
↪'zog_silicates', plotting=True)

# Now we delete the newly-created files. If you want them, comment out these␣
↪lines.
import os
os.remove('axisem_zog_silicates.txt')
os.remove('mineos_zog_silicates.txt')
```

```
Writing axisem_zog_silicates.txt ...
```




```
Writing mineos_zog_silicates.txt ...
```



Let's compare the properties of this planet to PREM

[11]:
```
import warnings
prem = burnman.seismic.PREM()
premdepth = prem.internal_depth_list()
premradii = 6371.e3 - premdepth

with warnings.catch_warnings(record=True) as w:
    eval = prem.evaluate(['density', 'pressure', 'gravity', 'v_s', 'v_p'])
    premdensity, prempressure, premgravity, premvs, premvp = eval
    print(w[-1].message)
```

```
Gravity is not given in PREM and is now being computed. This will only work when␣
↪density is defined for the entire planet. Use at your own risk.
```

Also create the Anzellini et al. (2013) geotherm:

[12]:
```
from scipy.interpolate import interp1d
d = np.loadtxt('../tutorial/data/Anzellini_2013_geotherm.dat')
Anz_interp = interp1d(d[:,0]*1.e9, d[:,1])
```

Finally, plot the 1D structure of the planet

[13]:
```
fig = plt.figure(figsize=(8, 5))
ax = [fig.add_subplot(2, 2, i) for i in range(1, 5)]


bounds = np.array([[layer.radii[0]/1.e3, layer.radii[-1]/1.e3]
```

(continues on next page)

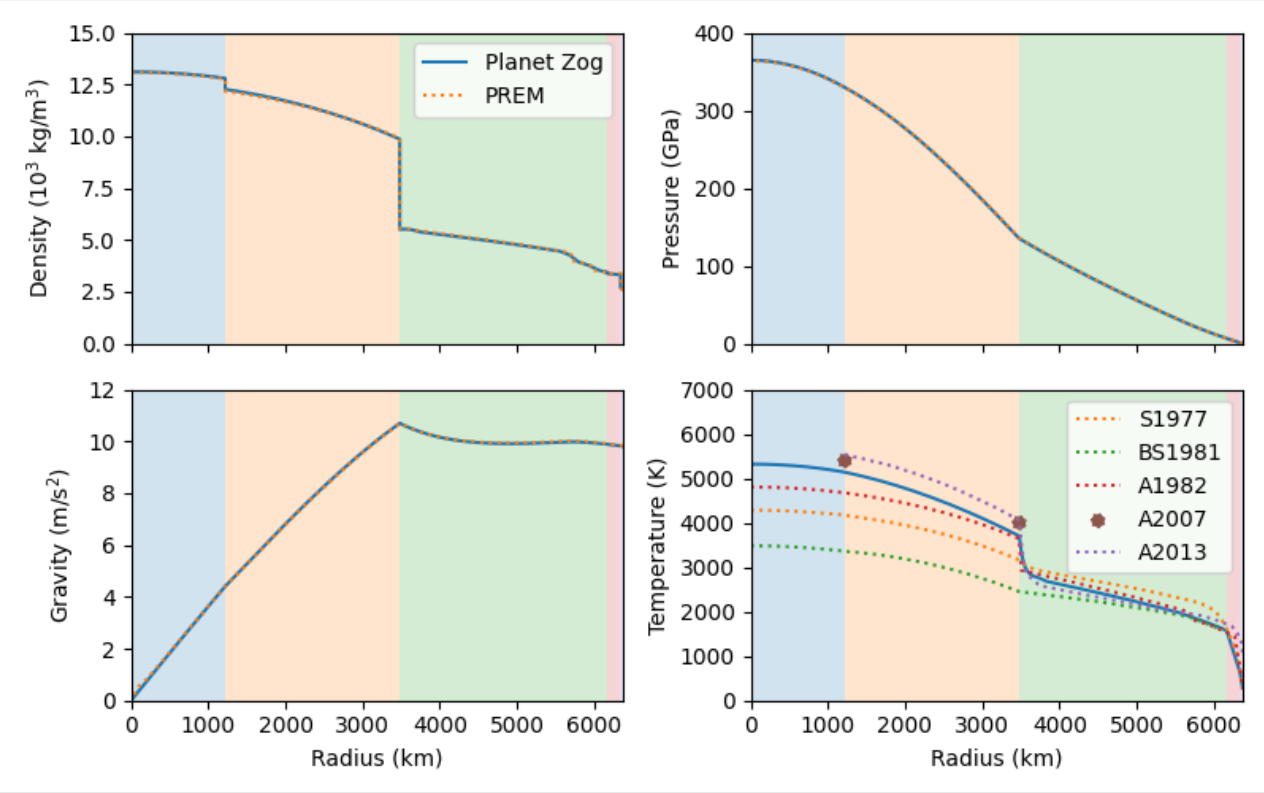
(continued from previous page)

```
                    for layer in planet_zog.layers])
maxy = [15, 400, 12, 7000]
for bound in bounds:
    for i in range(4):
        ax[i].fill_betweenx([0., maxy[i]],
                            [bound[0], bound[0]],
                            [bound[1], bound[1]], alpha=0.2)

ax[0].plot(planet_zog.radii / 1.e3, planet_zog.density / 1.e3,
           label=planet_zog.name)
ax[0].plot(premradii / 1.e3, premdensity / 1.e3, linestyle=':', label='PREM')
ax[0].set_ylabel('Density ($10^3$ kg/m$^3$)')
ax[0].legend()

# Make a subplot showing the calculated pressure profile
ax[1].plot(planet_zog.radii / 1.e3, planet_zog.pressure / 1.e9)
ax[1].plot(premradii / 1.e3, prempressure / 1.e9, linestyle=':')
ax[1].set_ylabel('Pressure (GPa)')

# Make a subplot showing the calculated gravity profile
ax[2].plot(planet_zog.radii / 1.e3, planet_zog.gravity)
ax[2].plot(premradii / 1.e3, premgravity, linestyle=':')
ax[2].set_ylabel('Gravity (m/s$^2)$')
ax[2].set_xlabel('Radius (km)')

# Make a subplot showing the calculated temperature profile
ax[3].plot(planet_zog.radii / 1.e3, planet_zog.temperature)
ax[3].set_ylabel('Temperature (K)')
ax[3].set_xlabel('Radius (km)')
ax[3].set_ylim(0.,)

# Finally, let's overlay some geotherms onto our model
# geotherm
labels = ['Stacey (1977)',
          'Brown and Shankland (1981)',
          'Anderson (1982)',
          'Alfe et al. (2007)',
          'Anzellini et al. (2013)']

short_labels = ['S1977',
                'BS1981',
                'A1982',
                'A2007',
                'A2013']

ax[3].plot(planet_zog.radii / 1.e3,
```

(continues on next page)

(continued from previous page)

```
burnman.geotherm.stacey_continental(planet_zog.depth),
linestyle=':', label=short_labels[0])
mask = planet_zog.depth > 269999.
ax[3].plot(planet_zog.radii[mask] / 1.e3,
           burnman.geotherm.brown_shankland(planet_zog.depth[mask]),
           linestyle=':', label=short_labels[1])
ax[3].plot(planet_zog.radii / 1.e3,
           burnman.geotherm.anderson(planet_zog.depth),
           linestyle=':', label=short_labels[2])

ax[3].scatter([planet_zog.layers[0].radii[-1] / 1.e3,
               planet_zog.layers[1].radii[-1] / 1.e3],
              [5400., 4000.],
              linestyle=':', label=short_labels[3])

mask = planet_zog.pressure < 330.e9
temperatures = Anz_interp(planet_zog.pressure[mask])
ax[3].plot(planet_zog.radii[mask] / 1.e3, temperatures,
           linestyle=':', label=short_labels[4])

ax[3].legend()

for i in range(2):
    ax[i].set_xticklabels([])
for i in range(4):
    ax[i].set_xlim(0., max(planet_zog.radii) / 1.e3)
    ax[i].set_ylim(0., maxy[i])

fig.set_tight_layout(True)
plt.show()
```



And that's it! Next time, we'll look at some of BurnMan's fitting routines.

The BurnMan Tutorial

3.4 Part 4: Fitting

This file is part of BurnMan - a thermoelastic and thermodynamic toolkit for the Earth and Planetary Sciences

Copyright (C) 2012 - 2021 by the BurnMan team, released under the GNU GPL v2 or later.

3.4.1 Introduction

This ipython notebook is the fourth in a series designed to introduce new users to the code structure and functionalities present in BurnMan.

Demonstrates

1. burnman.optimize.eos_fitting.fit_PTV_data
2. burnman.optimize.composition_fitting.fit_composition_to_solution
3. burnman.optimize.composition_fitting.fit_phase_proportions_to_bulk_composition

Everything in BurnMan and in this tutorial is defined in SI units.

[1]:
```
import burnman
import numpy as np
import matplotlib.pyplot as plt
```

```
Warning: No module named 'cdd'. For full functionality of BurnMan, please␣
↪install pycddlib.
```

3.4.2 Fitting parameters for an equation of state to experimental data

BurnMan contains least-squares optimization functions that fit model parameters to data. There are two helper functions especially for use in fitting Mineral parameters to experimental data; these are `burnman.optimize.eos_fitting.fit_PTp_data` (which can fit multiple kinds of data at the same time), and `burnman.optimize.eos_fitting.fit_PTV_data`, which specifically fits only pressure-temperature-volume data.

An extended example of fitting various kinds of data, outlier removal and detailed analysis can be found in `examples/example_fit_eos.py`. In this tutorial, we shall focus solely on fitting room temperature pressure-temperature-volume data. Specifically, the data we will fit is experimental volumes of stishovite, taken from Andrault et al. (2003). This data is provided in the form [P (GPa), V (Angstrom^3) and sigma_V (Angstrom^3)].

[2]:
```
PV = np.array([[0.0001, 46.5126, 0.0061],
               [1.168, 46.3429, 0.0053],
               [2.299, 46.1756, 0.0043],
               [3.137, 46.0550, 0.0051],
               [4.252, 45.8969, 0.0045],
               [5.037, 45.7902, 0.0053],
               [5.851, 45.6721, 0.0038],
               [6.613, 45.5715, 0.0050],
               [7.504, 45.4536, 0.0041],
               [8.264, 45.3609, 0.0056],
               [9.635, 45.1885, 0.0042],
               [11.69, 44.947, 0.002],
               [17.67, 44.264, 0.002],
               [22.38, 43.776, 0.003],
               [29.38, 43.073, 0.009],
               [37.71, 42.278, 0.008],
               [46.03, 41.544, 0.017],
               [52.73, 40.999, 0.009],
               [26.32, 43.164, 0.006],
               [30.98, 42.772, 0.005],
               [34.21, 42.407, 0.003],
               [38.45, 42.093, 0.004],
               [43.37, 41.610, 0.004],
               [47.49, 41.280, 0.007]])

print(f'{len(PV)} data points loaded successfully.')
```

```
24 data points loaded successfully.
```

BurnMan works exclusively in SI units, so we have to convert from GPa to Pa, and Angstrom per cell into molar volume in m^3. The fitting function also takes covariance matrices as input, so we have to build those matrices.

[3]:
```
from burnman.tools.unitcell import molar_volume_from_unit_cell_volume

Z = 2. # number of formula units per unit cell in stishovite
PTV = np.array([PV[:,0]*1.e9,
                298.15 * np.ones_like(PV[:,0]),
                molar_volume_from_unit_cell_volume(PV[:,1], Z)]).T

# Here, we assume that the pressure uncertainties are equal to 3% of the total␣
↪pressure,
# that the temperature uncertainties are negligible, and take the unit cell␣
↪volume
# uncertainties from the paper.
# We also assume that the uncertainties in pressure and volume are uncorrelated.
nul = np.zeros_like(PTV[:,0])
PTV_covariances = np.array([[0.03*PTV[:,0], nul, nul],
                            [nul, nul, nul],
                            [nul, nul, molar_volume_from_unit_cell_volume(PV[:,
↪2], Z)]]).T
PTV_covariances = np.power(PTV_covariances, 2.)
```

The next code block creates a Mineral object (`stv`), providing starting guesses for the parameters. The user selects which parameters they wish to fit, and which they wish to keep fixed. The parameters of the Mineral object are automatically updated during fitting.

Finally, the optimized parameter values and their variances are printed to screen.

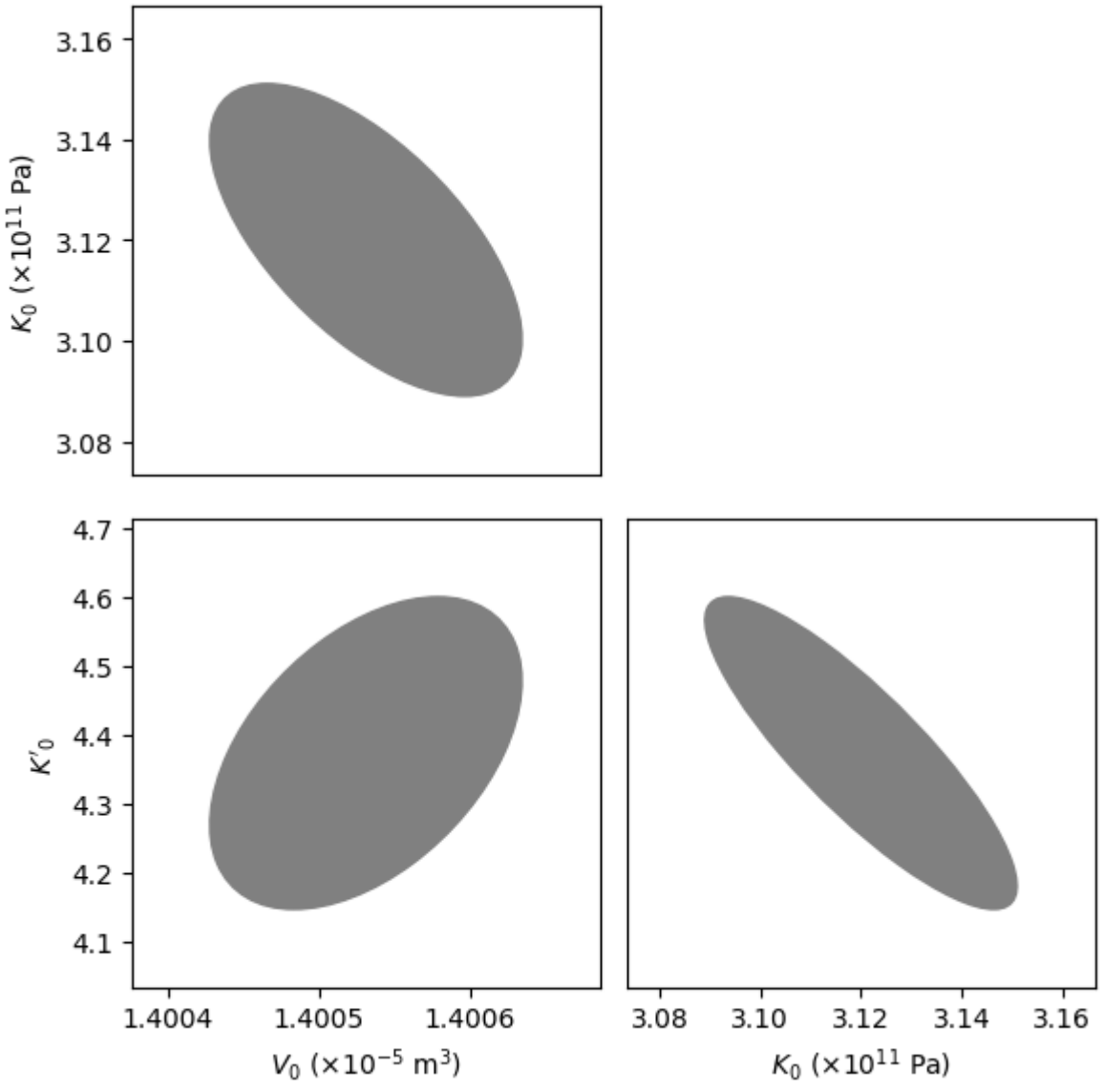
[4]:
```
stv = burnman.minerals.HP_2011_ds62.stv()
params = ['V_0', 'K_0', 'Kprime_0']
fitted_eos = burnman.optimize.eos_fitting.fit_PTV_data(stv, params, PTV, PTV_
↪covariances, verbose=False)

print('Optimized equation of state for stishovite:')
burnman.utils.misc.pretty_print_values(fitted_eos.popt, fitted_eos.pcov, fitted_
↪eos.fit_params)
print('')
```

```
Optimized equation of state for stishovite:
V_0: (1.4005 +/- 0.0001) x 1e-05
K_0: (3.12 +/- 0.03) x 1e+11
Kprime_0: (4.4 +/- 0.2) x 1e+00
```

The `fitted_eos` object contains a lot of useful information about the fit. In the next code block, we fit the

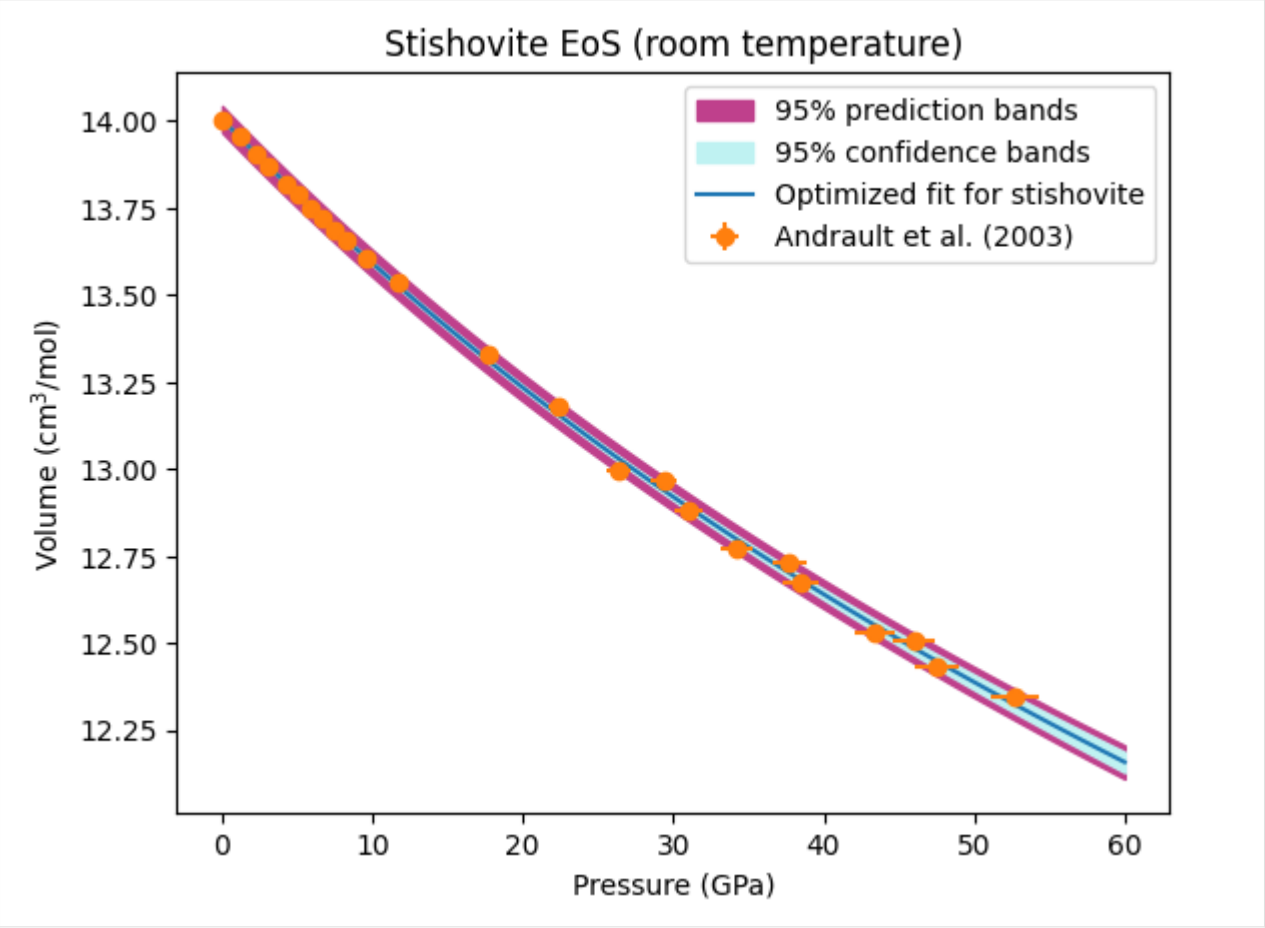
corner plot of the covariances, showing the tradeoffs in different parameters.

[5]:
```
import matplotlib
matplotlib.rc('axes.formatter', useoffset=False) # turns offset off, makes for a␣
↪more readable plot

fig = burnman.nonlinear_fitting.corner_plot(fitted_eos.popt, fitted_eos.pcov,
                                            params)
axes = fig[1]
axes[1][0].set_xlabel('$V_0$ ($\\times 10^{-5}$ m$^3$)')
axes[1][1].set_xlabel('$K_0$ ($\\times 10^{11}$ Pa)')
axes[0][0].set_ylabel('$K_0$ ($\\times 10^{11}$ Pa)')
axes[1][0].set_ylabel('$K\'_0$')
plt.show()
```



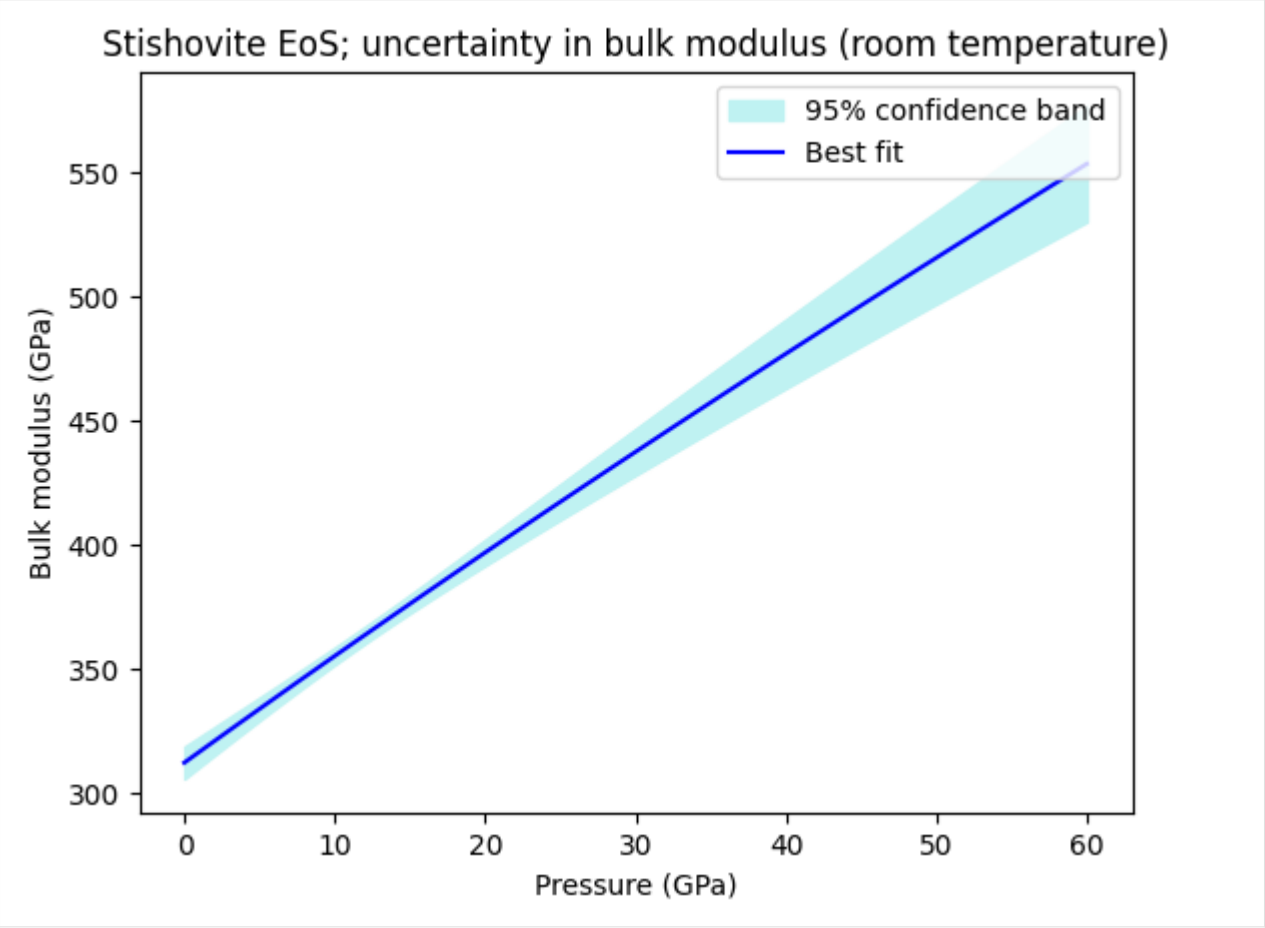
We now plot our optimized equation of state against the original data. BurnMan also includes a useful function `burnman.optimize.nonlinear_fitting.confidence_prediction_bands` that can be used to calculate the nth percentile confidence and prediction bounds on a function given a model using the delta method.

[6]:
```
from burnman.utils.misc import attribute_function
from burnman.optimize.nonlinear_fitting import confidence_prediction_bands

T = 298.15
pressures = np.linspace(1.e5, 60.e9, 101)
temperatures = T*np.ones_like(pressures)
volumes = stv.evaluate(['V'], pressures, temperatures)[0]
PTVs = np.array([pressures, temperatures, volumes]).T

# Calculate the 95% confidence and prediction bands
cp_bands = confidence_prediction_bands(model=fitted_eos,
                                       x_array=PTVs,
                                       confidence_interval=0.95,
                                       f=attribute_function(stv, 'V'),
                                       flag='V')

plt.fill_between(pressures/1.e9, cp_bands[2] * 1.e6, cp_bands[3] * 1.e6,
                 color=[0.75, 0.25, 0.55], label='95% prediction bands')
plt.fill_between(pressures/1.e9, cp_bands[0] * 1.e6, cp_bands[1] * 1.e6,
                 color=[0.75, 0.95, 0.95], label='95% confidence bands')

plt.plot(PTVs[:,0] / 1.e9, PTVs[:,2] * 1.e6, label='Optimized fit for stishovite
↪')
plt.errorbar(PTV[:,0] / 1.e9, PTV[:,2] * 1.e6,
             xerr=np.sqrt(PTV_covariances[:,0,0]) / 1.e9,
             yerr=np.sqrt(PTV_covariances[:,2,2]) * 1.e6,
             linestyle='None', marker='o', label='Andrault et al. (2003)')

plt.ylabel("Volume (cm$^3$/mol)")
plt.xlabel("Pressure (GPa)")
plt.legend(loc="upper right")
plt.title("Stishovite EoS (room temperature)")
plt.show()
```



We can also calculate the confidence and prediction bands for any other property of the mineral. In the code block below, we calculate and plot the optimized isothermal bulk modulus and its uncertainties.

[7]:
```
cp_bands = confidence_prediction_bands(model=fitted_eos,
                                       x_array=PTVs,
                                       confidence_interval=0.95,
                                       f=attribute_function(stv, 'K_T'),
                                       flag='V')

plt.fill_between(pressures/1.e9, (cp_bands[0])/1.e9, (cp_bands[1])/1.e9,␣
↪color=[0.75, 0.95, 0.95], label='95% confidence band')
plt.plot(pressures/1.e9, (cp_bands[0] + cp_bands[1])/2.e9, color='b', label=
↪'Best fit')

plt.ylabel("Bulk modulus (GPa)")
plt.xlabel("Pressure (GPa)")
plt.legend(loc="upper right")
plt.title("Stishovite EoS; uncertainty in bulk modulus (room temperature)")
plt.show()
```



3.4.3 Finding the best fit endmember proportions of a solution given a bulk composition

Let's now turn our focus to a different kind of fitting. It is common in petrology to have a bulk composition of a phase (provided, for example, by electron probe microanalysis), and want to turn this composition into a formula that satisfies stoichiometric constraints. This can be formulated as a constrained, weighted least squares problem, and BurnMan can be used to solve these problems using the function `burnman.optimize.composition_fitting.fit_composition_to_solution`.

In the following example, we shall create a model garnet composition, and then fit that to the Jennings and Holland (2015) garnet solution model. First, let's look at the solution model endmembers (pyrope, almandine, grossular, andradite and knorringite):

[8]:
```
from burnman import minerals

gt = minerals.JH_2015.garnet()

print(f'Endmembers: {gt.endmember_names}')
print(f'Elements: {gt.elements}')
```

(continues on next page)

(continued from previous page)

```
print('Stoichiometric matrix:')
print(gt.stoichiometric_matrix)
```

```
Endmembers: ['py', 'alm', 'gr', 'andr', 'knor']
Elements: ['Ca', 'Mg', 'Cr', 'Fe', 'Al', 'Si', 'O']
Stoichiometric matrix:
Matrix([[0, 3, 0, 0, 2, 3, 12], [0, 0, 0, 3, 2, 3, 12], [3, 0, 0, 0, 2, 3, 12],␣
↪[3, 0, 0, 2, 0, 3, 12], [0, 3, 2, 0, 0, 3, 12]])
```

Now, let's create a model garnet composition. A unique composition can be determined with the species Fe (total), Ca, Mg, Cr, Al, Si and Fe3+, all given in mole amounts. On top of this, we add some random noise (using a fixed seed so that the composition is reproducible).

[9]:
```
fitted_variables = ['Fe', 'Ca', 'Mg', 'Cr', 'Al', 'Si', 'Fe3+']
variable_values = np.array([1.1, 2., 0., 0, 1.9, 3., 0.1])
variable_covariances = np.eye(7)*0.01*0.01

# Add some noise.
v_err = np.random.rand(7)
np.random.seed(100)
variable_values = np.random.multivariate_normal(variable_values,
                                                variable_covariances)
```

Importantly, Fe3+ isn't an element or a site-species of the solution model, so we need to provide the linear conversion from Fe3+ to elements and/or site species. In this case, Fe3+ resides only on the second site (Site B), and the JH_2015.gt model has labelled Fe3+ on that site as Fef. Therefore, the conversion is simply Fe3+ = Fef_B.

[10]:
```
variable_conversions = {'Fe3+': {'Fef_B': 1.}}
```

Now we're ready to do the fitting. The following line is all that is required, and yields as output the optimized parameters, the corresponding covariance matrix and the residual.

[11]:
```
from burnman.optimize.composition_fitting import fit_composition_to_solution
popt, pcov, res = fit_composition_to_solution(gt,
                                              fitted_variables,
                                              variable_values,
                                              variable_covariances,
                                              variable_conversions)
```

Finally, the optimized parameters can be used to set the composition of the solution model, and the optimized parameters printed to stdout.

[12]:
```
# We can set the composition of gt using the optimized parameters
gt.set_composition(popt)

# Print the optimized parameters and principal uncertainties
```

(continues on next page)

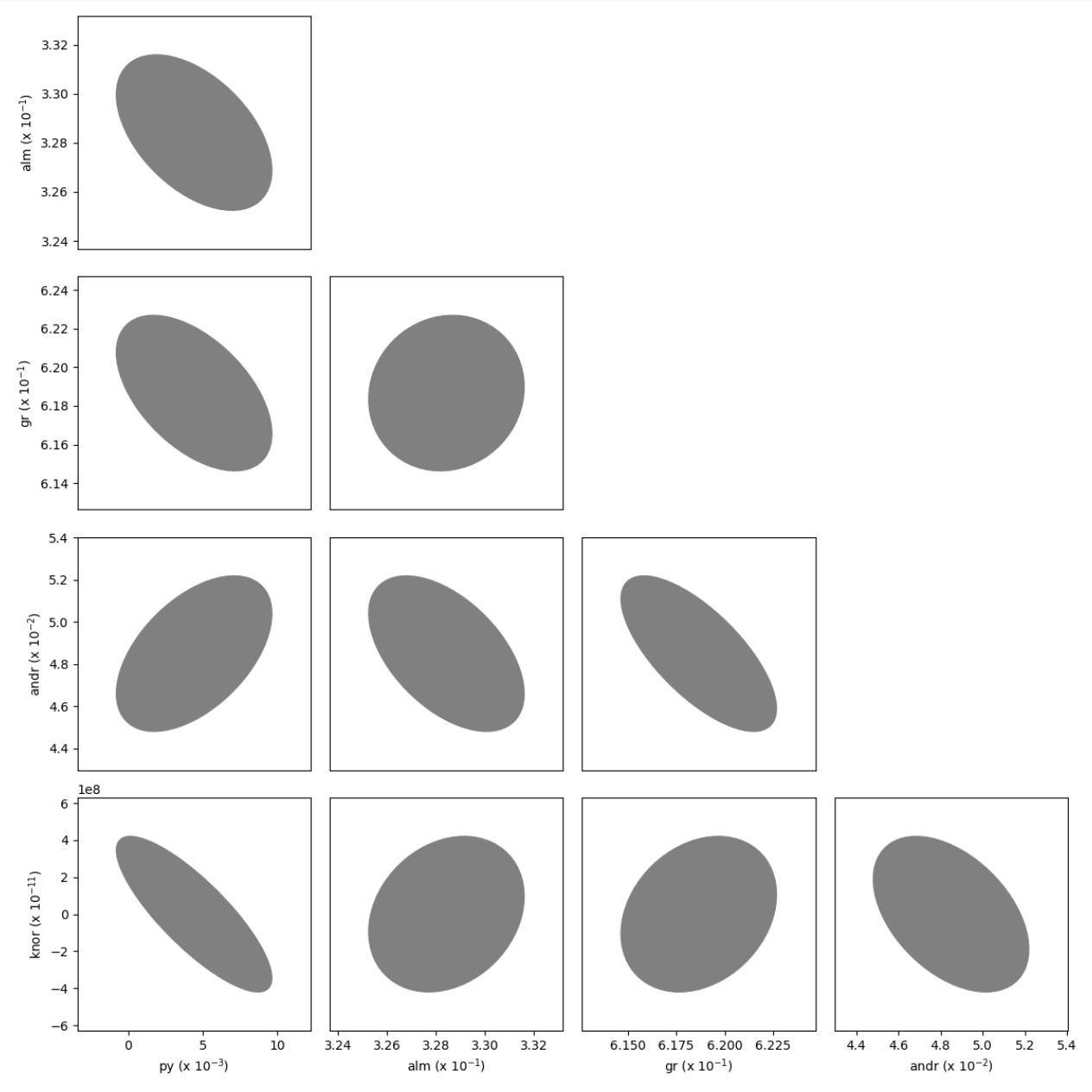
(continued from previous page)

```
print('Molar fractions:')
for i in range(len(popt)):
    print(f'{gt.endmember_names[i]}: '
            f'{gt.molar_fractions[i]:.3f} +/- '
            f'{np.sqrt(pcov[i][i]):.3f}')

print(f'Weighted residual: {res:.3f}')
```

```
Molar fractions:
py: 0.004 +/- 0.005
alm: 0.328 +/- 0.003
gr: 0.619 +/- 0.004
andr: 0.048 +/- 0.004
knor: 0.000 +/- 0.004
Weighted residual: 0.519
```

As in the equation of state fitting, a corner plot of the covariances can also be plotted.

[13]:
```
fig = burnman.nonlinear_fitting.corner_plot(popt, pcov, gt.endmember_names)
```



3.4.4 Fitting phase proportions to a bulk composition

Another common constrained weighted least squares problem involves fitting phase proportions, given their individual compositions and the overall bulk composition. This is particularly important in experimental petrology, where the bulk composition is known from a starting composition. In these cases, the residual after fitting is often used to assess whether the sample remained a closed system during the experiment.

In the following example, we take phase compositions and the bulk composition reported from high pressure experiments on a Martian mantle composition by Bertka and Fei (1997), and use these to calculate phase proportions in Mars mantle, and the quality of the experiments.

First, some tedious data preparation...

[14]:
```
import itertools

# Load and transpose input data
filename = '../burnman/data/input_fitting/Bertka_Fei_1997_mars_mantle.dat'
with open(filename) as f:
    column_names = f.readline().strip().split()[1:]
data = np.genfromtxt(filename, dtype=None, encoding='utf8')
data = list(map(list, itertools.zip_longest(*data, fillvalue=None)))

# The first six columns are compositions given in weight % oxides
compositions = np.array(data[:6])

# The first row is the bulk composition
bulk_composition = compositions[:, 0]

# Load all the data into a dictionary
data = {column_names[i]: np.array(data[i])
        for i in range(len(column_names))}

# Make ordered lists of samples (i.e. experiment ID) and phases
samples = []
phases = []
for i in range(len(data['sample'])):
    if data['sample'][i] not in samples:
        samples.append(data['sample'][i])
    if data['phase'][i] not in phases:
        phases.append(data['phase'][i])

samples.remove("bulk_composition")
phases.remove("bulk")

# Get the indices of all the phases present in each sample
sample_indices = [[i for i in range(len(data['sample']))
                   if data['sample'][i] == sample]
                   for sample in samples]

# Get the run pressures of each experiment
pressures = np.array([data['pressure'][indices[0]] for indices in sample_
↪indices])
```

The following code block loops over each of the compositions, and finds the best weight proportions and uncertainties on those proportions.

[15]:
```
from burnman.optimize.composition_fitting import fit_phase_proportions_to_bulk_
↪composition
```

(continues on next page)

(continued from previous page)

```
# Create empty arrays to store the weight proportions of each phase,
# and the principal uncertainties (we do not use the covariances here,
# although they are calculated)
weight_proportions = np.zeros((len(samples), len(phases)))*np.NaN
weight_proportion_uncertainties = np.zeros((len(samples),
                                            len(phases)))*np.NaN

residuals = []
# Loop over the samples, fitting phase proportions
# to the provided bulk composition
for i, sample in enumerate(samples):
    # This line does the heavy lifting
    popt, pcov, res = fit_phase_proportions_to_bulk_composition(compositions[:,␣
↪sample_indices[i]],
                                                                bulk_composition)

    residuals.append(res)

    # Fill the correct elements of the weight_proportions
    # and weight_proportion_uncertainties arrays
    sample_phases = [data['phase'][i] for i in sample_indices[i]]
    for j, phase in enumerate(sample_phases):
        weight_proportions[i, phases.index(phase)] = popt[j]
        weight_proportion_uncertainties[i, phases.index(phase)] = np.
↪sqrt(pcov[j][j])
```

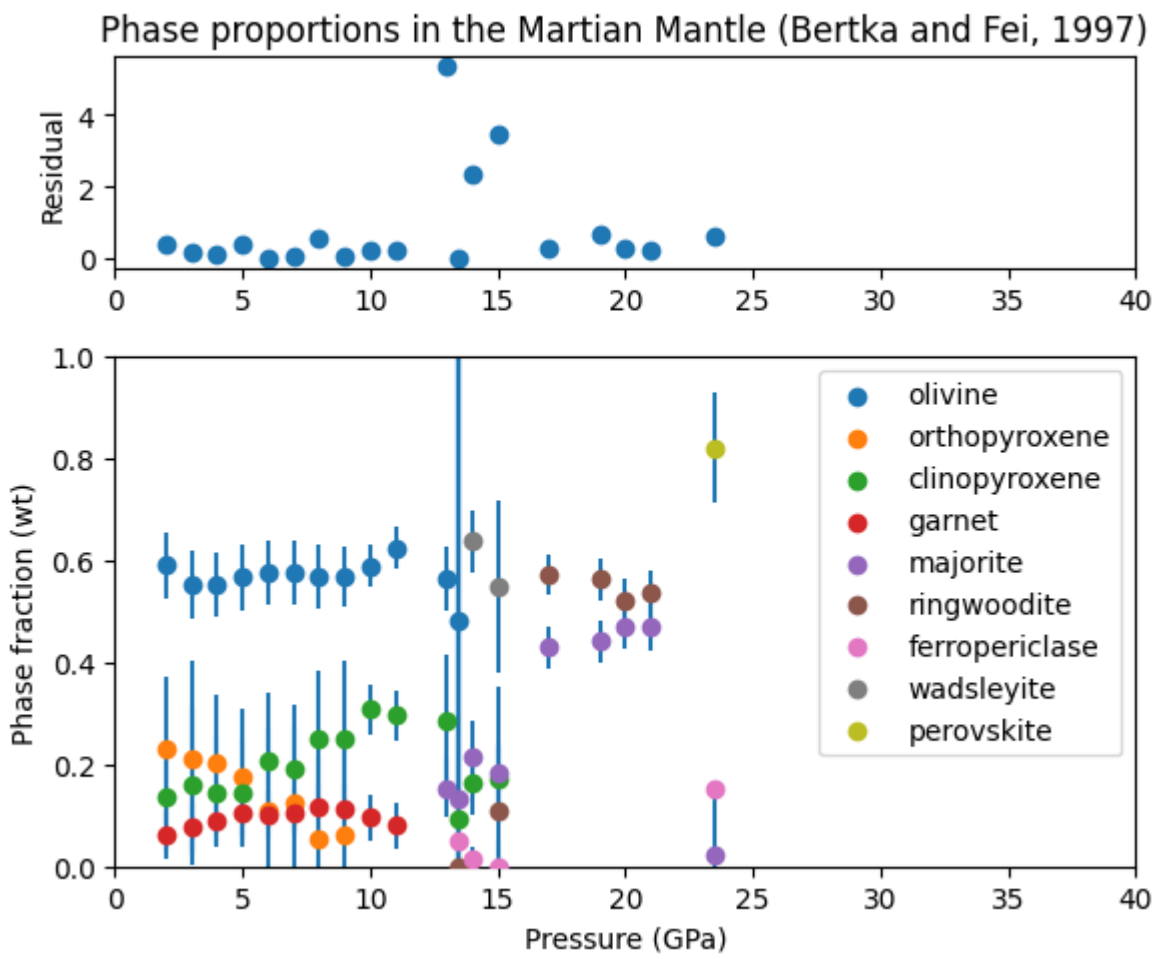
Finally, we plot the data.

[16]:
```
fig = plt.figure(figsize=(6, 5))
ax = [fig.add_subplot(3, 1, 1)]
ax.append(fig.add_subplot(3, 1, (2, 4)))
for i, phase in enumerate(phases):
    ebar = plt.errorbar(pressures, weight_proportions[:, i],
                        yerr=weight_proportion_uncertainties[:, i],
                        fmt="none", zorder=2)
    ax[1].scatter(pressures, weight_proportions[:, i], label=phase, zorder=3)

ax[0].set_title('Phase proportions in the Martian Mantle (Bertka and Fei, 1997)')
ax[0].scatter(pressures, residuals)
for i in range(2):
    ax[i].set_xlim(0., 40.)

ax[1].set_ylim(0., 1.)
ax[0].set_ylabel('Residual')
ax[1].set_xlabel('Pressure (GPa)')
```

(continues on next page)

(continued from previous page)

```
ax[1].set_ylabel('Phase fraction (wt)')
ax[1].legend()
fig.set_tight_layout(True)
plt.show()
```



We can see from this plot that most of the residuals are below one, indicating that the probe analyses consistent with the bulk composition. Three analyses have higher residuals, which may indicate a problem with the experiments, or with the analyses around the wadsleyite field.

The phase proportions also show some nice trends; clinopyroxene weight percentage increases with pressure at the expense of orthopyroxene. Garnet / majorite percentage increases sharply as clinopyroxene is exhausted at 14-16 GPa.

And we're done! Next time, we'll look at how to determine equilibrium assemblages in BurnMan.

The BurnMan Tutorial

3.5 Part 5: Equilibrium problems

3.5.1 Introduction

This ipython notebook is the fifth in a series designed to introduce new users to the code structure and functionalities present in BurnMan.

Demonstrates

1. burnman.equilibrate, an experimental function that determines the bulk elemental composition, pressure, temperature, phase proportions and compositions of an assemblage subject to user-defined constraints.

Everything in BurnMan and in this tutorial is defined in SI units.

3.5.2 Phase equilibria

3.5.2.1 What BurnMan does and doesn't do

Members of the BurnMan Team are often asked whether BurnMan does Gibbs energy minimization. The short answer to that is no, for three reasons: 1) Python is ill-suited to such computationally intensive problems. 2) There are many pieces of software already in the community that do Gibbs energy minimization, including but not limited to: PerpleX, HeFESTo, Theriak Domino, MELTS, ENKI, FactSAGE (proprietary), and MMA-EoS. 3) Gibbs minimization is a hard problem. The brute-force pseudocompound/simplex technique employed by Perple_X is the only globally robust method, but clever techniques have to be used to make the computations tractable, and the solution found is generally only a (very close) approximation to the true minimum assemblage. More refined Newton / higher order schemes (e.g. HeFESTo, MELTS, ENKI) provide an exact solution, but can get stuck in local minima or even fail to find a solution.

So, with those things in mind, what *does* BurnMan do? Well, because BurnMan can compute the Gibbs energy and analytical derivatives of composite materials, it is well suited to solving the equilibrium relations for *fixed assemblages*. This is done using the `burnman.equilibrate` function, which acts in a similar (but slightly more general) way to the THERMOCALC software developed by Tim Holland, Roger Powell and coworkers. Essentially, one chooses an assemblage (e.g. olivine + garnet + orthopyroxene) and some equality constraints (typically related to bulk composition, pressure, temperature, entropy, volume, phase proportions or phase compositions) and the `equilibrate` function attempts to find the remaining unknowns that satisfy those constraints.

In a sense, then, the `equilibrate` function is simultaneously more powerful and more limited than Gibbs minimization techniques. It allows the user to investigate and plot metastable reactions, and quickly obtain answers to questions like "at what pressure does wadsleyite first become stable along a given isentrope?". However, it is not designed to create P-T tables of equilibrium assemblages. If a user wishes to do this for a complex problem, we refer them to other existing codes. BurnMan also contains a useful utility material called `burnman.PerplexMaterial` that is specifically designed to read in and interrogate P-T data from PerpleX.

There are a couple more caveats to bear in mind. Firstly, the `equilibrate` function is experimental and can certainly be improved. Equilibrium problems are highly nonlinear, and sometimes solvers struggle to find a solution. If you have a better, more robust way of solving these problems, we would *love* to hear from you! Secondly, the `equilibrate` function is not completely free from the curse of multiple roots - sometimes there is more than one solution to the equilibrium problem, and BurnMan (and indeed any equilibrium software) may find one a metastable root.

3.5.3 Equilibrating at fixed bulk composition

Fixed bulk composition problems are most similar to those asked by Gibbs minimization software like HeFESTo. Essentially, the only difference is that rather than allowing the assemblage to change to minimize the Gibbs energy, the assemblage is instead fixed.

In the following code block, we calculate the equilibrium assemblage of olivine, orthopyroxene and garnet for a mantle composition in the system NCFMAS at 10 GPa and 1500 K.

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import burnman
from burnman import equilibrate
from burnman.minerals import SLB_2011

# Set the pressure, temperature and composition
pressure = 3.e9
temperature = 1500.
composition = {'Na': 0.02, 'Fe': 0.2, 'Mg': 2.0, 'Si': 1.9,
               'Ca': 0.2, 'Al': 0.4, 'O': 6.81}

# Create the assemblage
gt = SLB_2011.garnet()
ol = SLB_2011.mg_fe_olivine()
opx = SLB_2011.orthopyroxene()
assemblage = burnman.Composite(phases=[ol, opx, gt],
                               fractions=[0.7, 0.1, 0.2],
                               name='NCFMAS ol-opx-gt assemblage')

# The solver uses the current compositions of each solution as a starting guess,
# so we have to set them here
ol.set_composition([0.93, 0.07])
opx.set_composition([0.8, 0.1, 0.05, 0.05])
gt.set_composition([0.8, 0.1, 0.05, 0.03, 0.02])

equality_constraints = [('P', 10.e9), ('T', 1500.)]

sol, prm = equilibrate(composition, assemblage, equality_constraints)

print(f'It is {sol.success} that equilibrate was successful')
```

(continues on next page)

(continued from previous page)

```
print(sol.assemblage)

# The total entropy of the assemblage is the molar entropy
# multiplied by the number of moles in the assemblage
entropy = sol.assemblage.S*sol.assemblage.n_moles
```

```
Warning: No module named 'cdd'. For full functionality of BurnMan, please␣
↪install pycddlib.
It is True that equilibrate was successful
Composite: NCFMAS ol-opx-gt assemblage
  P, T: 1e+10 Pa, 1500 K
Phase and endmember fractions:
  olivine: 0.4971
    Forsterite: 0.9339
    Fayalite: 0.0661
  orthopyroxene: 0.2925
    Enstatite: 0.8640
    Ferrosilite: 0.0687
    Mg_Tschermaks: 0.0005
    Ortho_Diopside: 0.0668
  garnet: 0.2104
    Pyrope: 0.4458
    Almandine: 0.1239
    Grossular: 0.2607
    Mg_Majorite: 0.1258
    Jd_Majorite: 0.0437
```

Each equality constraint can be a list of constraints, in which case equilibrate will loop over them. In the next code block we change the equality constraints to be a series of pressures which correspond to the total entropy obtained from the previous solve.

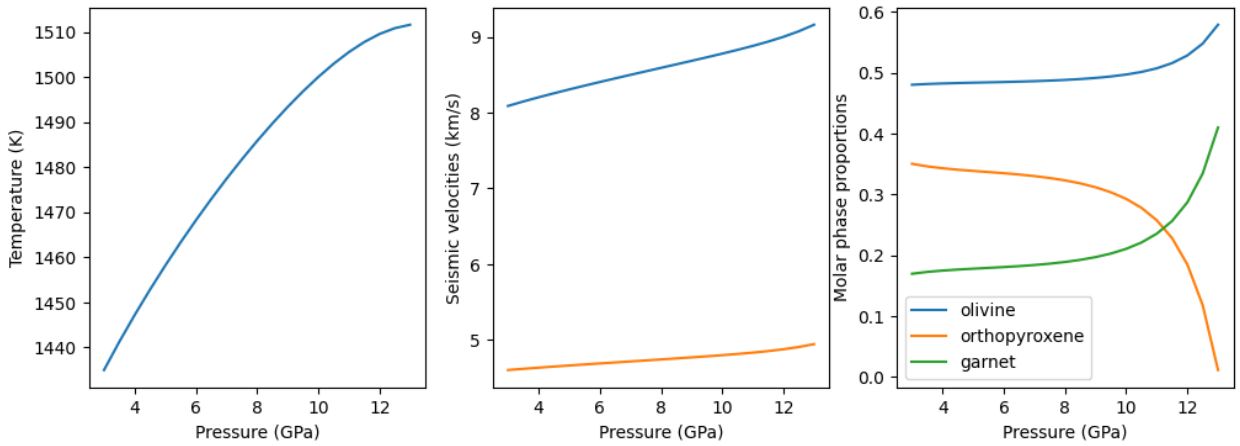
[2]:
```
equality_constraints = [('P', np.linspace(3.e9, 13.e9, 21)),
                        ('S', entropy)]

sols, prm = equilibrate(composition, assemblage, equality_constraints)
```

The object `sols` is now a 1D list of solution objects. Each one of these contains an equilibrium assemblage object that can be interrogated for any properties:

[3]:
```
data = np.array([[sol.assemblage.pressure,
                  sol.assemblage.temperature,
                  sol.assemblage.p_wave_velocity,
                  sol.assemblage.shear_wave_velocity,
                  sol.assemblage.molar_fractions[0],
                  sol.assemblage.molar_fractions[1],
                  sol.assemblage.molar_fractions[2]]
                 for sol in sols if sol.success])
```

The next code block plots these properties.

[4]:
```
fig = plt.figure(figsize=(12, 4))
ax = [fig.add_subplot(1, 3, i) for i in range(1, 4)]

P, T, V_p, V_s = data.T[:4]
phase_proportions = data.T[4:]
ax[0].plot(P/1.e9, T)
ax[1].plot(P/1.e9, V_p/1.e3)
ax[1].plot(P/1.e9, V_s/1.e3)

for i in range(3):
    ax[2].plot(P/1.e9, phase_proportions[i], label=sol.assemblage.phases[i].name)

for i in range(3):
    ax[i].set_xlabel('Pressure (GPa)')
ax[0].set_ylabel('Temperature (K)')
ax[1].set_ylabel('Seismic velocities (km/s)')
ax[2].set_ylabel('Molar phase proportions')
ax[2].legend()

plt.show()
```



From the above figure, we can see that the proportion of orthopyroxene is decreasing rapidly and is exhausted near 13 GPa. In the next code block, we determine the exact pressure at which orthopyroxene is exhausted.

[5]:
```
equality_constraints = [('phase_fraction', [opx, 0.]),
                        ('S', entropy)]
sol, prm = equilibrate(composition, assemblage, equality_constraints)

print(f'Orthopyroxene is exhausted from the assemblage at {sol.assemblage.
↪pressure/1.e9:.2f} GPa, {sol.assemblage.temperature:.2f} K.')
```

```
Orthopyroxene is exhausted from the assemblage at 13.04 GPa, 1511.64 K.
```

3.5.4 Equilibrating while allowing bulk composition to vary

[6]:
```
# Initialize the minerals we will use in this example.
ol = SLB_2011.mg_fe_olivine()
wad = SLB_2011.mg_fe_wadsleyite()
rw = SLB_2011.mg_fe_ringwoodite()

# Set the starting guess compositions for each of the solutions
ol.set_composition([0.90, 0.10])
wad.set_composition([0.90, 0.10])
rw.set_composition([0.80, 0.20])
```

First, we find the compositions of the three phases at the univariant.

[7]:
```
T = 1600.
composition = {'Fe': 0.2, 'Mg': 1.8, 'Si': 1.0, 'O': 4.0}
assemblage = burnman.Composite([ol, wad, rw], [1., 0., 0.])
equality_constraints = [('T', T),
                        ('phase_fraction', (ol, 0.0)),
                        ('phase_fraction', (rw, 0.0))]
free_compositional_vectors = [{'Mg': 1., 'Fe': -1.}]

sol, prm = equilibrate(composition, assemblage, equality_constraints,
                       free_compositional_vectors,
                       verbose=False)
if not sol.success:
    raise Exception('Could not find solution for the univariant using '
                    'provided starting guesses.')

P_univariant = sol.assemblage.pressure
phase_names = [sol.assemblage.phases[i].name for i in range(3)]
x_fe_mbr = [sol.assemblage.phases[i].molar_fractions[1] for i in range(3)]

print(f'Univariant pressure at {T:.0f} K: {P_univariant/1.e9:.3f} GPa')
print('Fe2SiO4 concentrations at the univariant:')
for i in range(3):
    print(f'{phase_names[i]}: {x_fe_mbr[i]:.2f}')
```

```
Univariant pressure at 1600 K: 12.002 GPa
Fe2SiO4 concentrations at the univariant:
olivine: 0.22
wadsleyite: 0.37
ringwoodite: 0.50
```

Now we solve for the stable sections of the three binary loops

[8]:
```
output = []
for (m1, m2, x_fe_m1) in [[ol, wad, np.linspace(x_fe_mbr[0], 0.001, 20)],
```

(continues on next page)

(continued from previous page)

```
                          [ol, rw, np.linspace(x_fe_mbr[0], 0.999, 20)],
                          [wad, rw, np.linspace(x_fe_mbr[1], 0.001, 20)]]:

    assemblage = burnman.Composite([m1, m2], [1., 0.])

    # Reset the compositions of the two phases to have compositions
    # close to those at the univariant point
    m1.set_composition([1.-x_fe_mbr[1], x_fe_mbr[1]])
    m2.set_composition([1.-x_fe_mbr[1], x_fe_mbr[1]])

    # Also set the pressure and temperature
    assemblage.set_state(P_univariant, T)

    # Here our equality constraints are temperature,
    # the phase fraction of the second phase,
    # and we loop over the composition of the first phase.
    equality_constraints = [('T', T),
                            ('phase_composition',
                             (m1, [['Mg_A', 'Fe_A'],
                                   [0., 1.], [1., 1.], x_fe_m1])),
                            ('phase_fraction', (m2, 0.0))]

    sols, prm = equilibrate(composition, assemblage,
                            equality_constraints,
                            free_compositional_vectors,
                            verbose=False)

    # Process the solutions
    out = np.array([[sol.assemblage.pressure,
                     sol.assemblage.phases[0].molar_fractions[1],
                     sol.assemblage.phases[1].molar_fractions[1]]
                    for sol in sols if sol.success])
    output.append(out)

output = np.array(output)
```

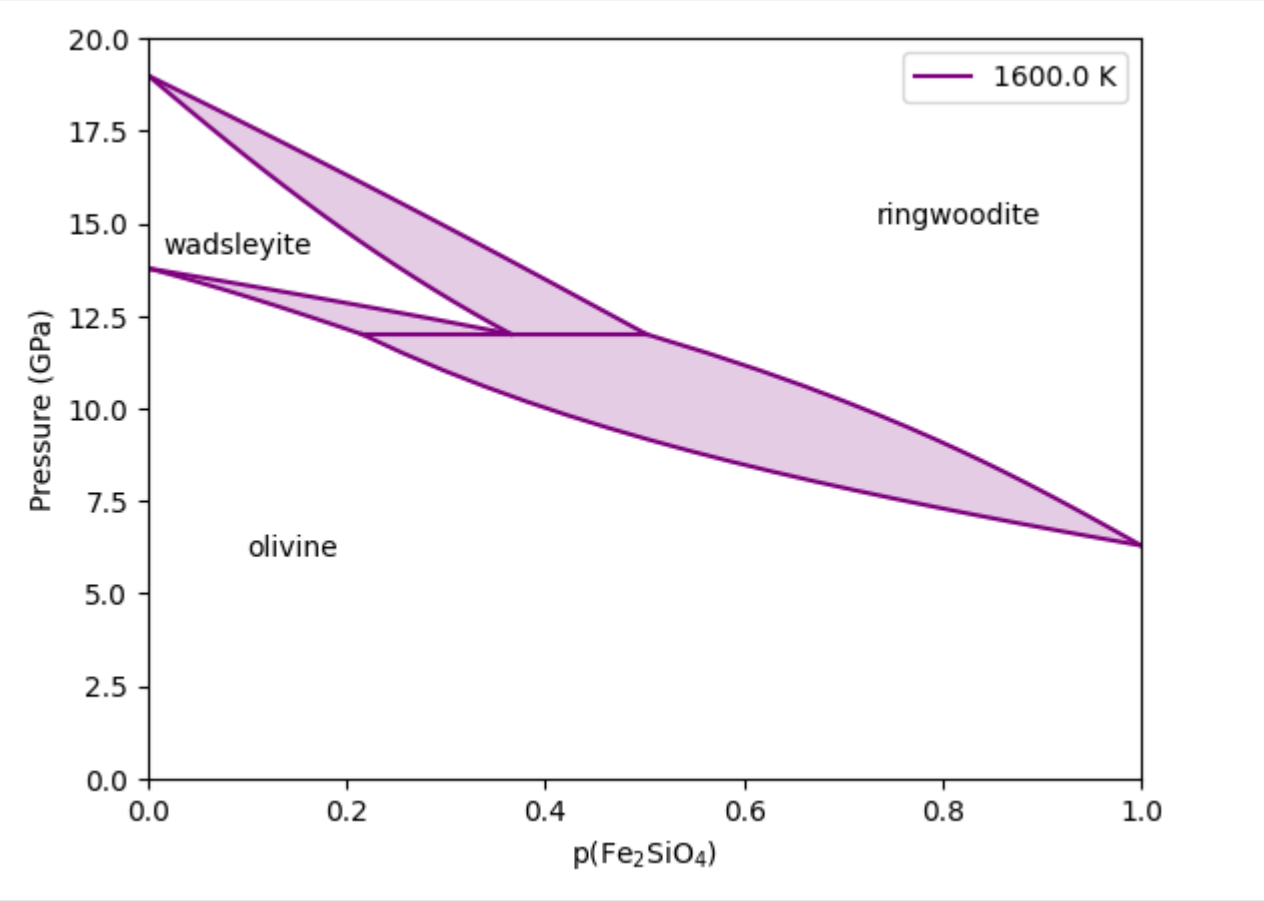
Finally, we do some plotting

[9]:
```
fig = plt.figure()
ax = [fig.add_subplot(1, 1, 1)]

color='purple'
# Plot the line connecting the three phases
ax[0].plot([x_fe_mbr[0], x_fe_mbr[2]],
           [P_univariant/1.e9, P_univariant/1.e9], color=color)
```

(continues on next page)

(continued from previous page)

```
for i in range(3):
    if i == 0:
        ax[0].plot(output[i,:,1], output[i,:,0]/1.e9, color=color, label=f'{T} K
↪')
    else:
        ax[0].plot(output[i,:,1], output[i,:,0]/1.e9, color=color)

    ax[0].plot(output[i,:,2], output[i,:,0]/1.e9, color=color)
    ax[0].fill_betweenx(output[i,:,0]/1.e9, output[i,:,1], output[i,:,2],
                        color=color, alpha=0.2)

ax[0].text(0.1, 6., 'olivine', horizontalalignment='left')
ax[0].text(0.015, 14.2, 'wadsleyite', horizontalalignment='left',
        bbox=dict(facecolor='white',
                  edgecolor='white',
                  boxstyle='round,pad=0.2'))
ax[0].text(0.9, 15., 'ringwoodite', horizontalalignment='right')

ax[0].set_xlim(0., 1.)
ax[0].set_ylim(0.,20.)
ax[0].set_xlabel('p(Fe$_2$SiO$_4$)')
ax[0].set_ylabel('Pressure (GPa)')
ax[0].legend()
plt.show()
```



And we're done!

CHAPTER

FOUR

CIDER TUTORIAL 2014

The tutorial for BurnMan presented at CIDER 2014 consists of three separate units:

- *step 1*,
- *step 2*, and
- *step 3*.

4.1 CIDER 2014 BurnMan Tutorial — step 1

In this first part of the tutorial we will acquaint ourselves with a basic script for calculating the elastic properties of a mantle mineralogical model.

In general, there are three portions of this script:

1) Define a set of pressures and temperatures at which we want to calculate elastic properties

2) Setup a composite of minerals (or "rock") and calculate its elastic properties at those pressures and temperatures.

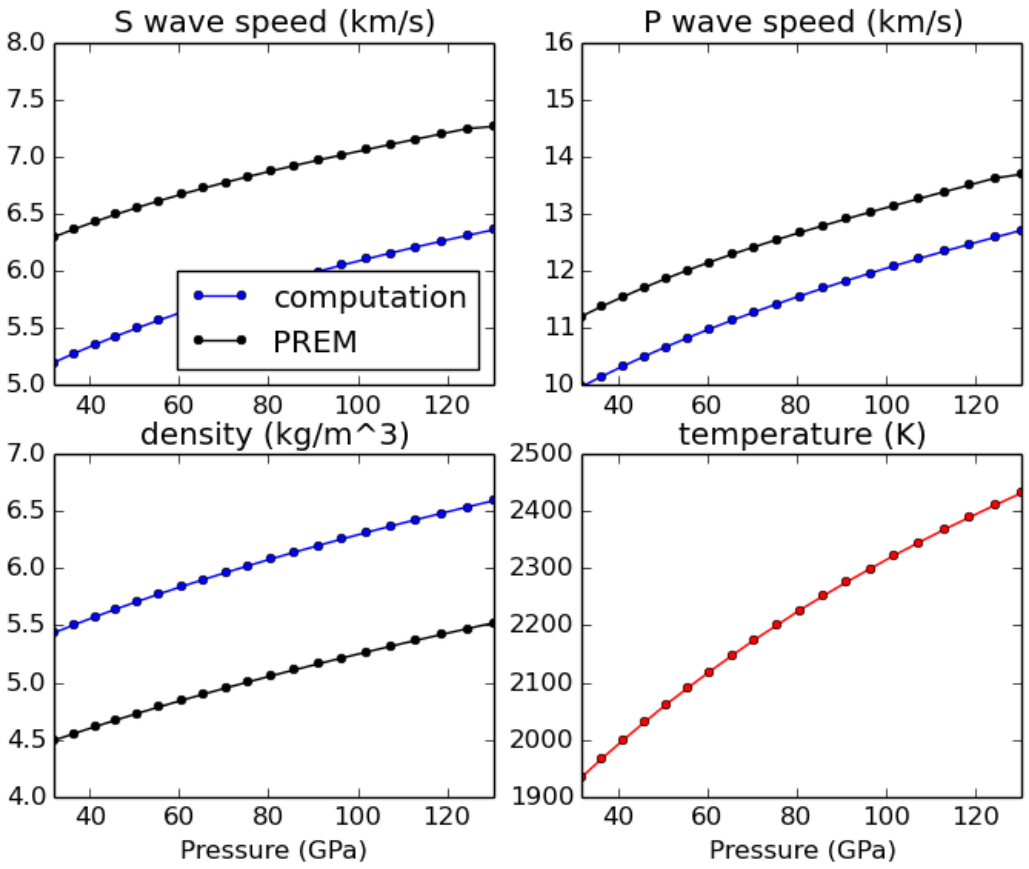
3) Plot those elastic properties, and compare them to a seismic model, in this case PREM

The script is basically already written, and should run as is by typing:

```
python step_1.py
```

on the command line. However, the mineral model for the rock is not very realistic, and you will want to change it to one that is more in accordance with what we think the bulk composition of Earth's lower mantle is.

When run (without putting in a more realistic composition), the program produces the following image:



Your goal in this tutorial is to improve this awful fit…

4.2 CIDER 2014 BurnMan Tutorial — step 2

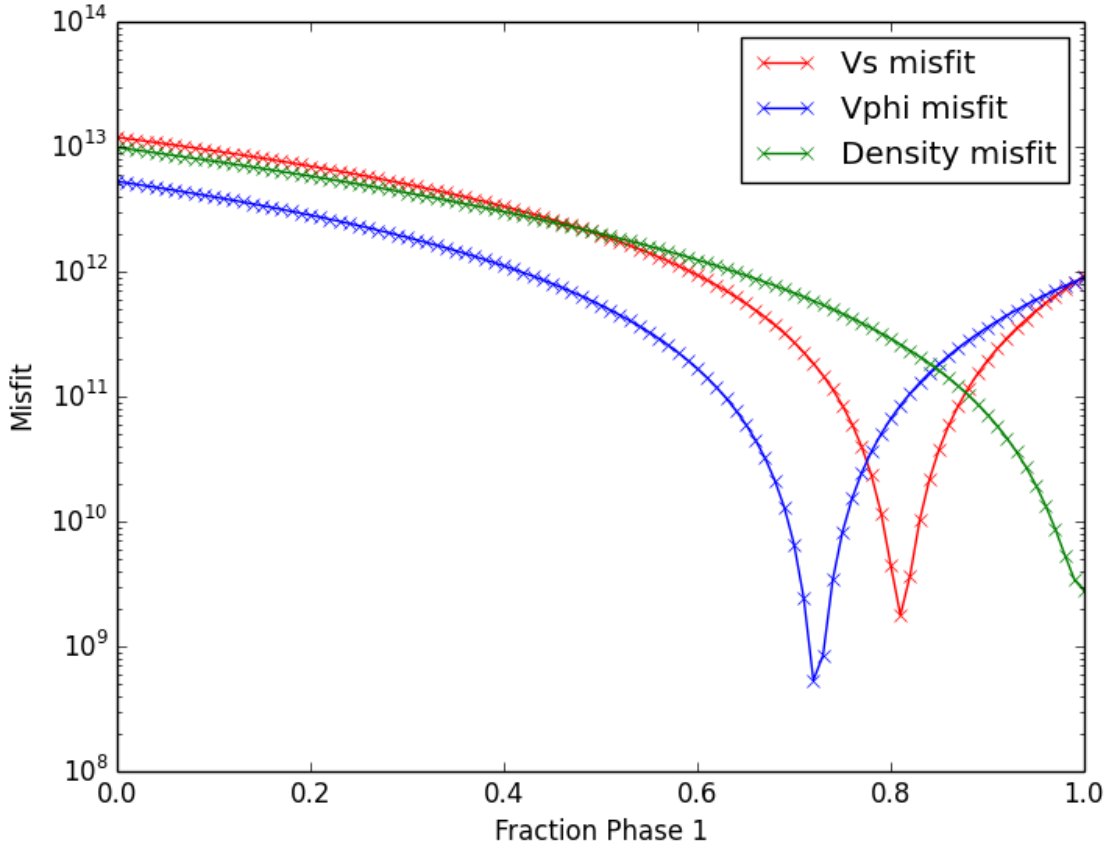
In this second part of the tutorial we try to get a closer fit to our 1D seismic reference model. In the simple Mg, Si, and O model that we used in step 1 there was one free parameter, namely phase_1_fraction, which goes between zero and one.

In this script we want to explore how good of a fit to PREM we can get by varying this fraction. We create a simple function that calculates a misfit between PREM and our mineral model as a function of phase_1_fraction, and then plot this misfit function to try to find a best model.

This script may be run by typing

```
python step_2.py
```

Without changing any input, the program should produce the following image showing the misfit as a function of perovskite content:



4.3 CIDER 2014 BurnMan Tutorial — step 3

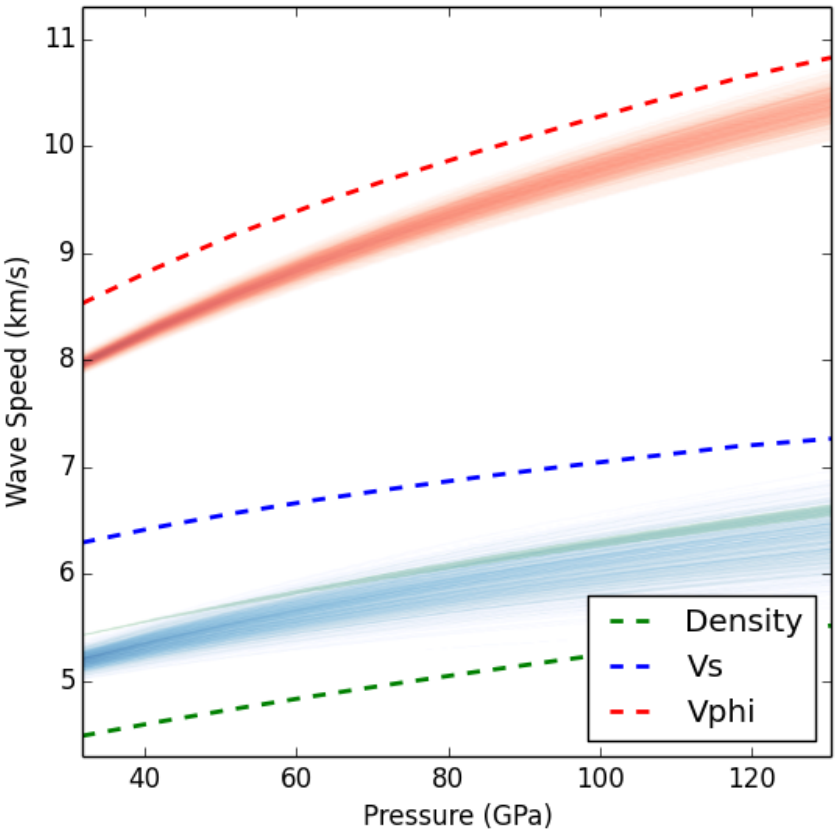
In the previous two steps of the tutorial we tried to find a very simple mineralogical model that best fit the 1D seismic model PREM. But we know that there is consideral uncertainty in many of the mineral physical parameters that control how the elastic properties of minerals change with pressure and temperature. In this step we explore how uncertainties in these parameters might affect the conclusions you draw.

The strategy here is to make many different "realizations" of the rock that you determined was the closest fit to PREM, where each realization has its mineral physical parameters perturbed by a small amount, hopefully related to the uncertainty in that parameter. In particular, we will look at how perturbations to K_0' and G_0' (the pressure derivatives of the bulk and shear modulus, respectively) change the calculated 1D seismic profiles.

This script may be run by typing

```
python step_3.py
```

After changing the standard deviations for K_0' and G_0' to 0.2, the following figure of velocities for 1000 realizations is produced:


11
10
9
8
7
6
5
Wave Speed (km/s)
40
60
80
100
120
Pressure (GPa)
Density
Vs
Vphi

CHAPTER

FIVE

EXAMPLES

BurnMan comes with a large collection of example programs under examples/. Below you can find a summary of the different examples. They are grouped into three categories: *Class examples*, *Simple Examples* and *More Advanced Examples*. The *Class examples* introduce the main class types in BurnMan, and covers similar ground to *The BurnMan Tutorial* but in a little more detail. *Simple Examples* provides introductions to some of the seismic and chemical functions. *More Advanced Examples* covers functionality which is more suited to research projects.

Finally, we also include the scripts that were used for all computations and figures in the 2014 BurnMan paper in the misc/ folder, see *Reproducing Cottaar, Heister, Rose and Unterborn (2014)*.

5.1 Class examples

The following is a list of examples that introduce the main classes of BurnMan objects:

- *example_mineral*,
- *example_gibbs_modifiers*,
- *example_solution*,
- *example_composite*,
- *example_calibrants*,
- *example_anisotropy*,
- *example_anisotropic_mineral*,
- *example_geotherms*, and
- *example_composition*.

5.1.1 example_mineral

This example shows how to create mineral objects in BurnMan, and how to output their thermodynamic and thermoelastic quantities.

Mineral objects are the building blocks for more complex objects in BurnMan. These objects are intended to represent minerals (or melts, or fluids) of fixed composition, with a well defined equation of state that defines the relationship between the current state (pressure and temperature) of the mineral and its thermodynamic potentials and derivatives (such as volume and entropy).

Mineral objects are initialized with a dictionary containing all of the parameters required by the desired equation of state. BurnMan contains implementations of may equations of state (*Equations of state*).

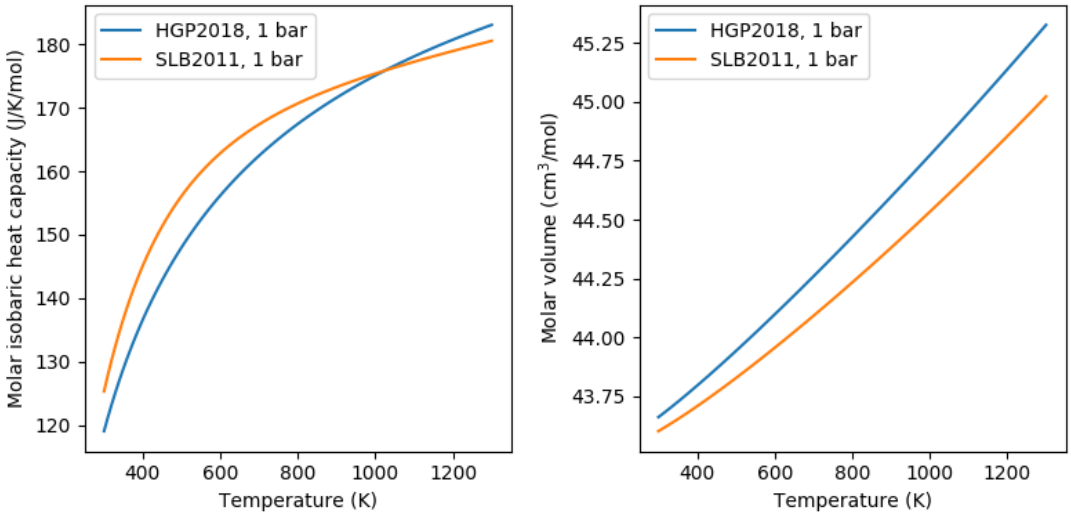
Uses:

- *Mineral databases*
- `burnman.Mineral`

Demonstrates:

- Different ways to define an endmember
- How to set state
- How to output thermodynamic and thermoelastic properties

Resulting figure:



5.1.2 example_gibbs_modifiers

This example script demonstrates the modifications to the gibbs free energy (and derivatives) that can be applied as masks over the results from the equations of state.

These modifications currently take the forms:

- Landau corrections (implementations of Putnis (1992) and Holland and Powell (2011)
- Bragg-Williams corrections (implementation of Holland and Powell (1996))
- Linear (a simple delta_E + delta_V*P - delta_S*T
- Magnetic (Chin, Hertzman and Sundman (1987))

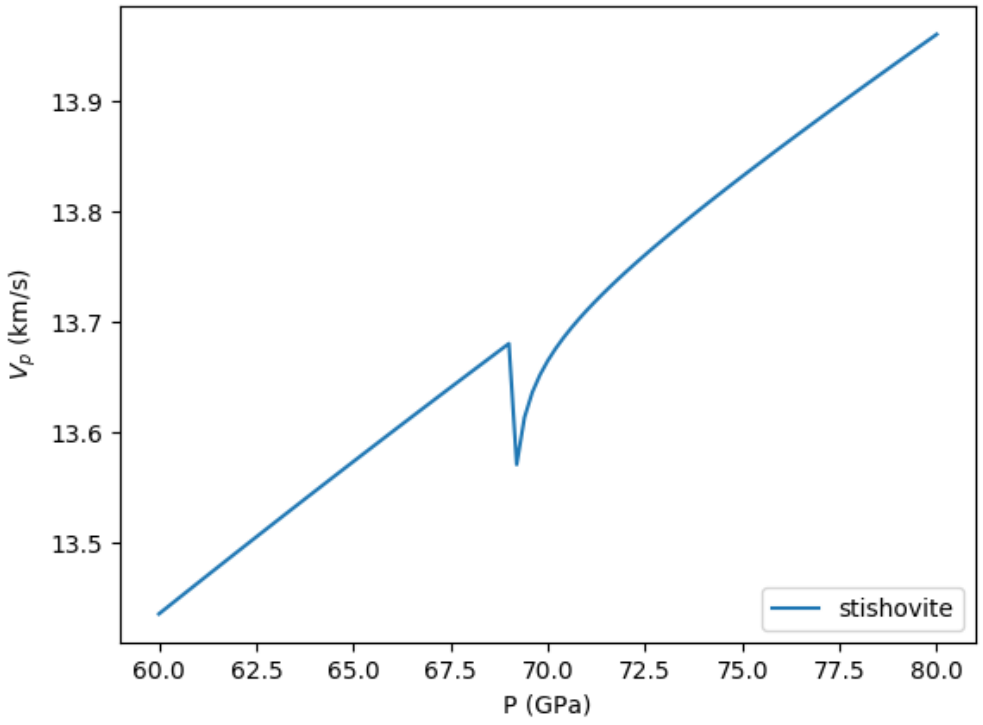
Uses:

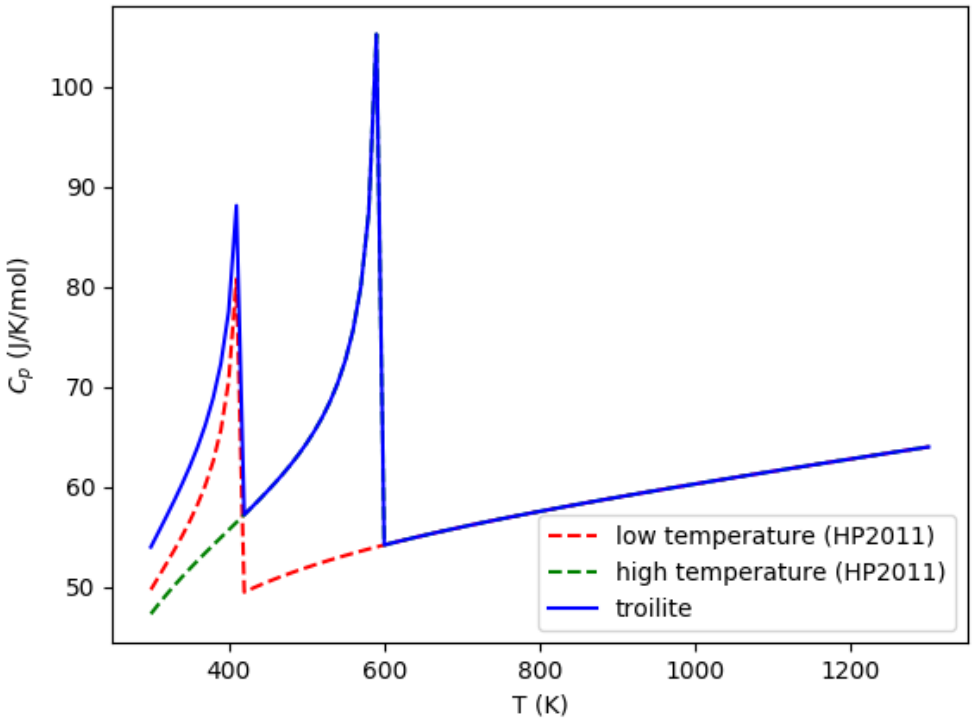
- *Mineral databases*

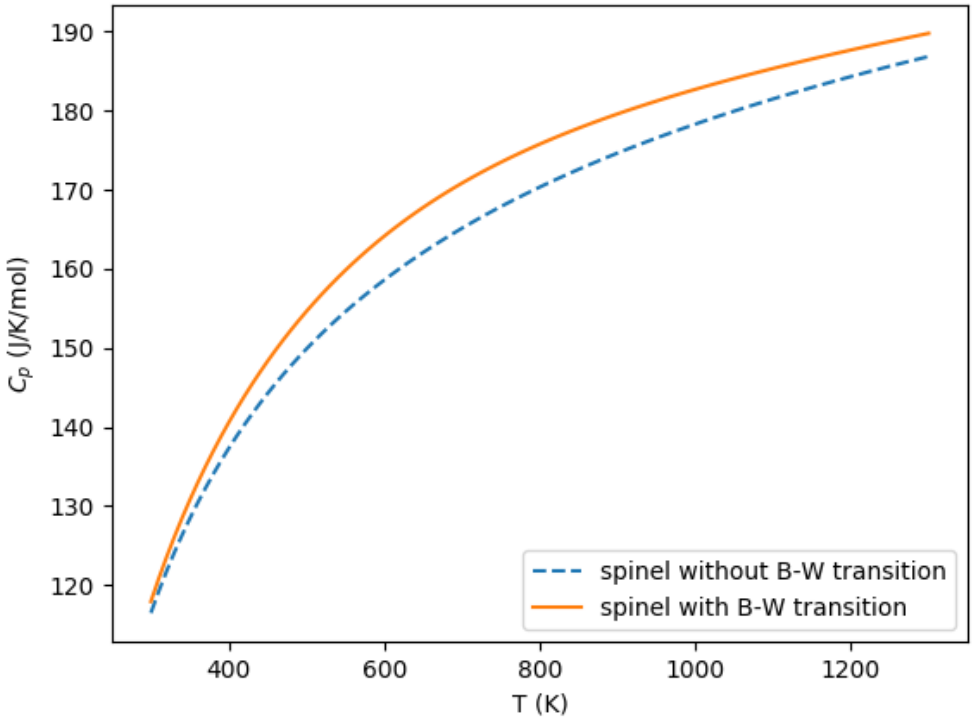
Demonstrates:

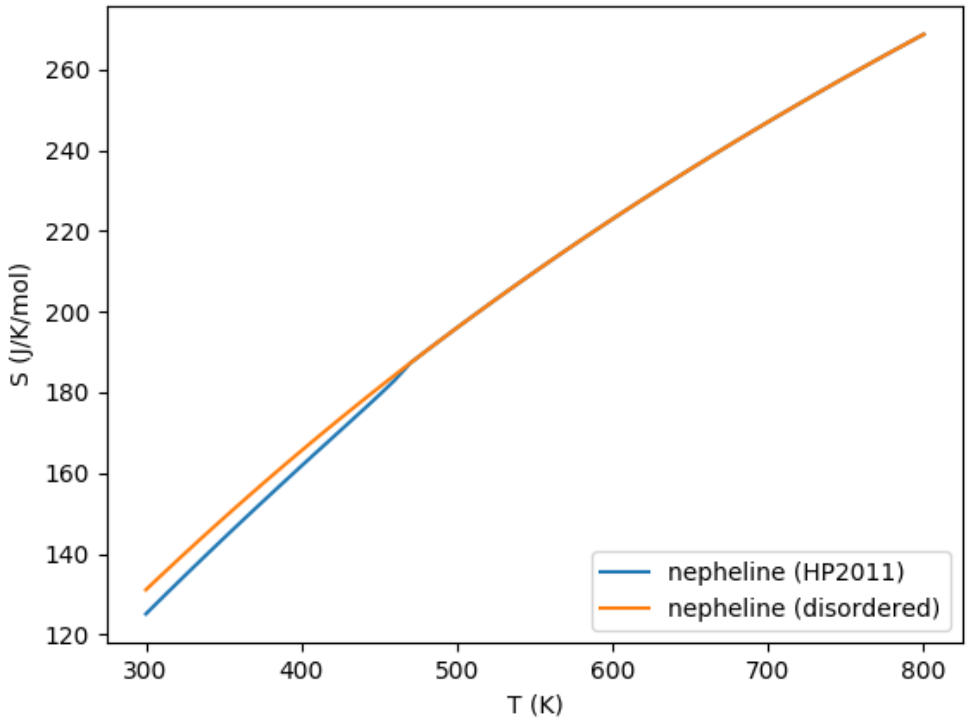
- creating a mineral with excess contributions
- calculating thermodynamic properties

Resulting figures:




100
90
80
70
60
50
C_p (J/K/mol)
400
600
800
1000
1200
T (K)
low temperature (HP2011)
high temperature (HP2011)
troilite


190
180
170
160
150
140
130
120
C_p (J/K/mol)
400
600
800
1000
1200
T (K)
spinel without B-W transition
spinel with B-W transition



5.1.3 example_solution

This example shows how to create different solution models and output thermodynamic and thermoelastic quantities.

There are four main types of solution currently implemented in BurnMan:

1. Ideal solutions
2. Symmmetric solutions
3. Asymmetric solutions
4. Subregular solutions

These solutions can potentially deal with:

- Disordered endmembers (more than one element on a crystallographic site)
- Site vacancies
- More than one valence/spin state of the same element on a site

Uses:

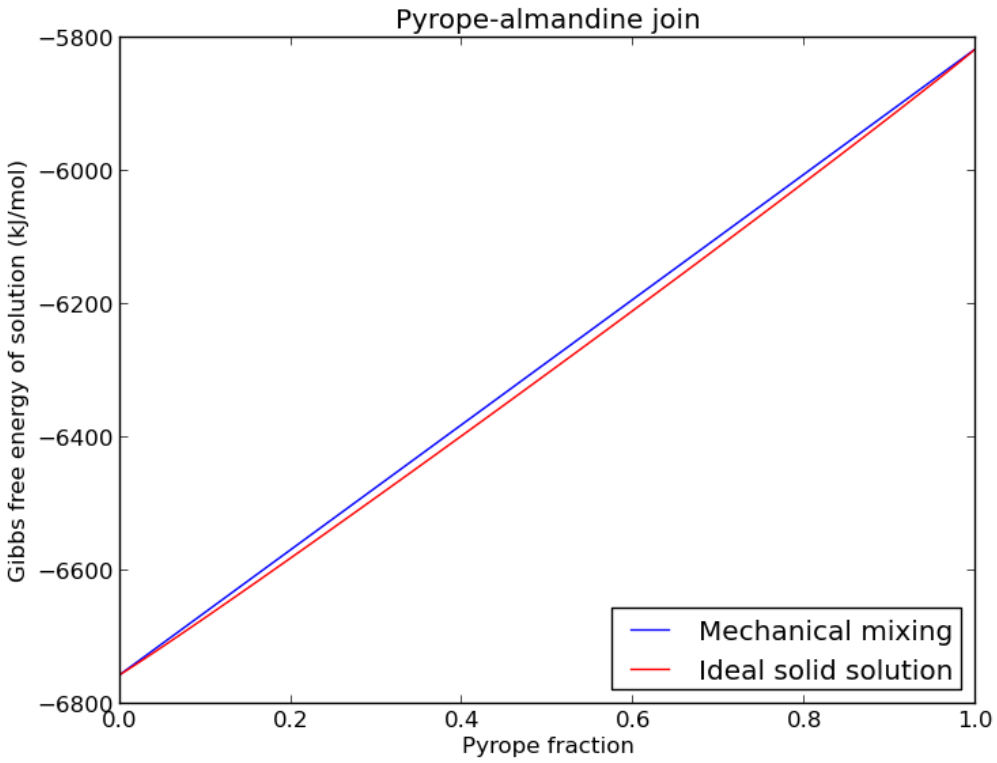
- *Mineral databases*
- `burnman.Solution`

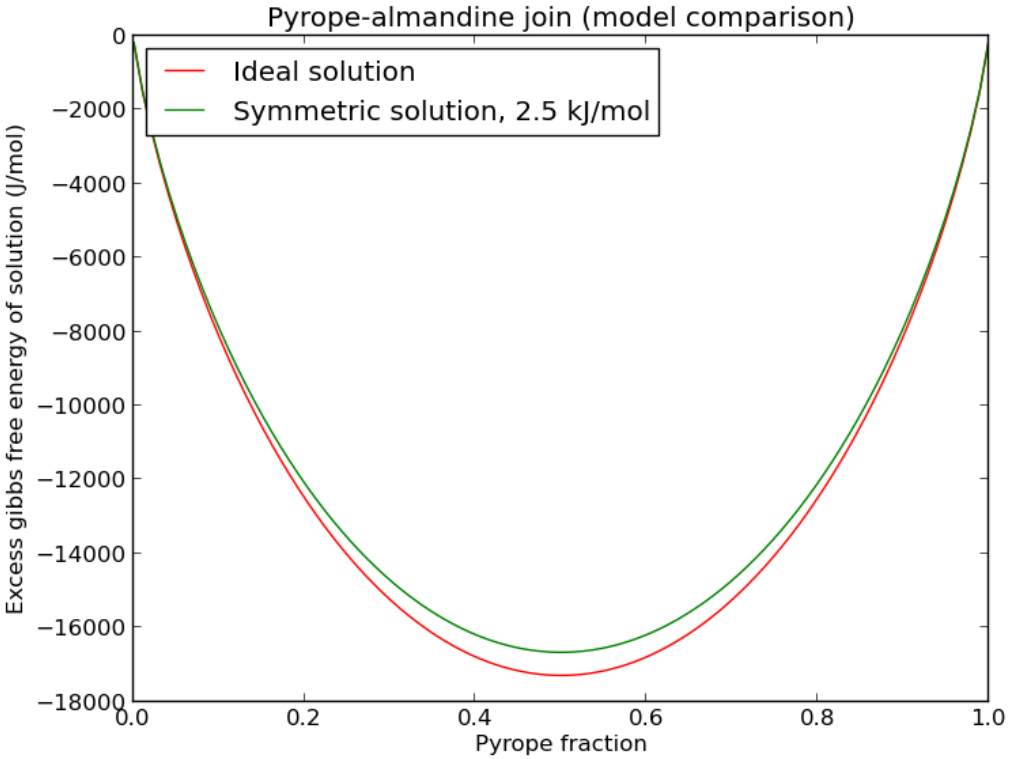
- `burnman.SolutionModel`

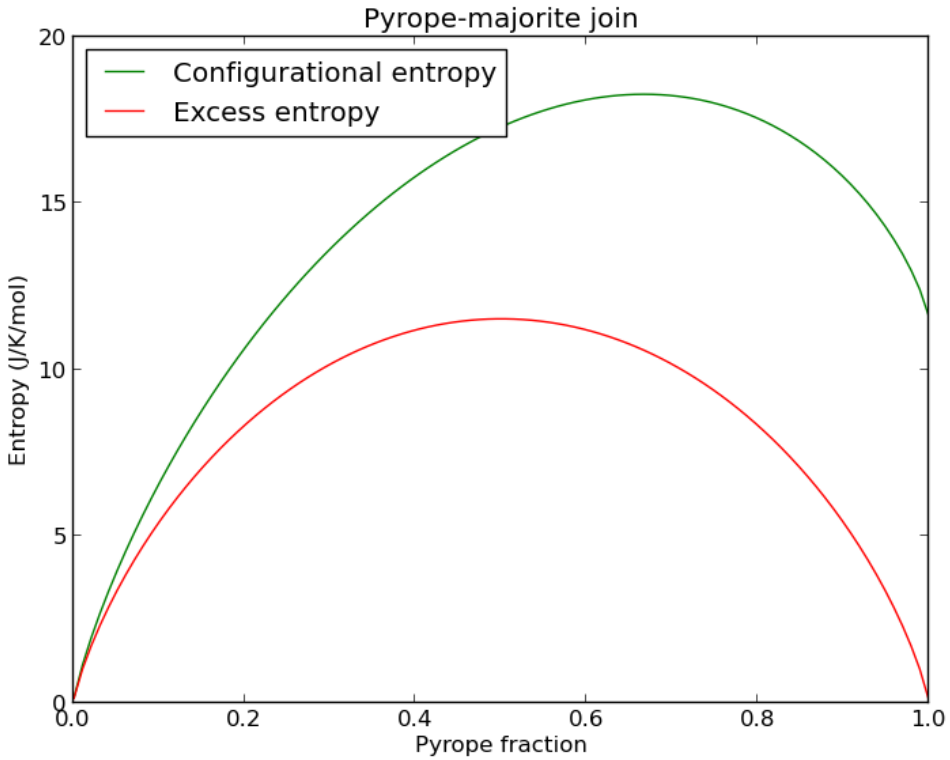
Demonstrates:

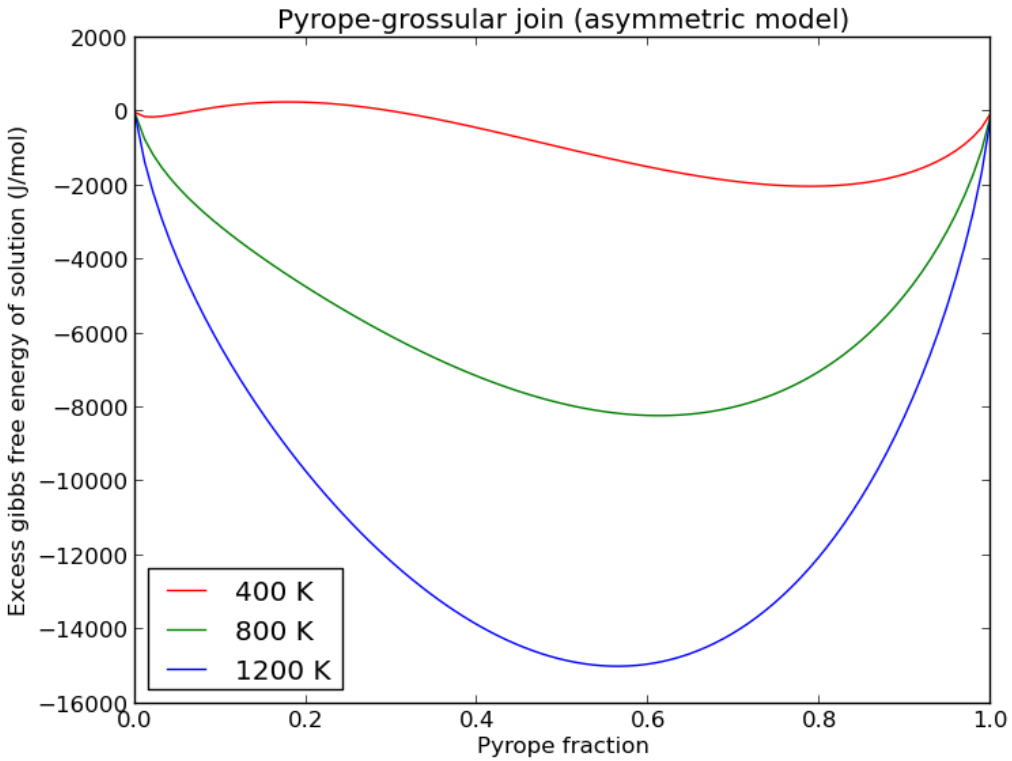
- Different ways to define a solution
- How to set composition and state
- How to output thermodynamic and thermoelastic properties

Resulting figures:




Pyrope-almandine join (model comparison)
Ideal solution
Symmetric solution, 2.5 kJ/mol
Excess gibbs free energy of solution (J/mol)
0
−2000
−4000
−6000
−8000
−10000
−12000
−14000
−16000
−18000
0.0
0.2
0.4
0.6
0.8
1.0
Pyrope fraction


Pyrope-majorite join
Configurational entropy
Excess entropy
Entropy (J/K/mol)
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
Pyrope fraction



5.1.4 example_composite

This example demonstrates the functionalities of the burnman.Composite class.

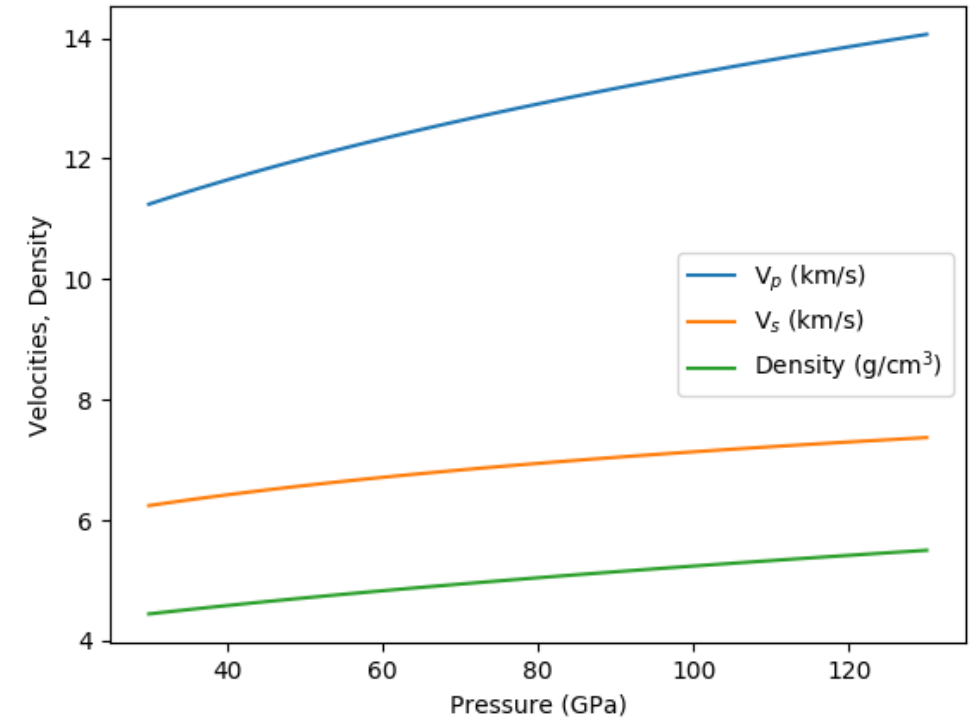
Uses:

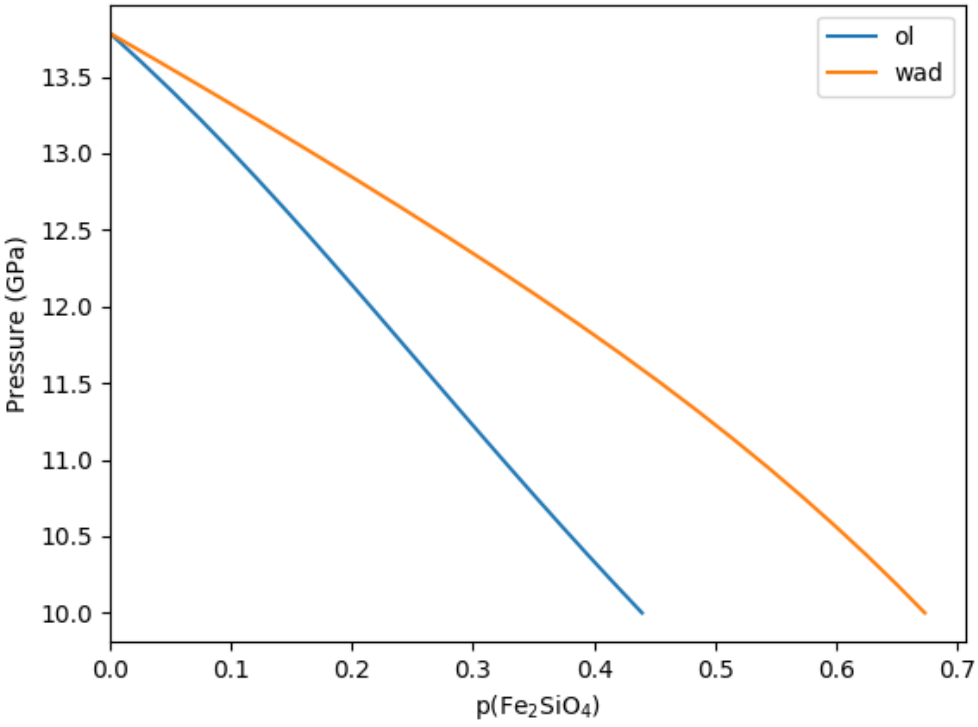
- *Mineral databases*
- `burnman.Mineral`
- `burnman.Solution`
- `burnman.Composite`

Demonstrates:

- How to initialize a composite object containing minerals and solutions
- How to set state and composition of composite objects
- How to interrogate composite objects for their compositional, thermodynamic and thermoelastic properties.
- How to use the stoichiometric and reaction affinity methods to solve simple thermodynamic equilibrium problems.

Resulting figures:


14
12
10
8
6
4
Velocities, Density
40
60
80
100
120
Pressure (GPa)
V_p (km/s)
V_s (km/s)
Density (g/cm^3)


13.5
13.0
12.5
12.0
11.5
11.0
10.5
10.0
Pressure (GPa)
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
p(Fe_2SiO_4)
ol
wad

5.1.5 example_mineral

This example shows how to create mineral objects in BurnMan, and how to output their thermodynamic and thermoelastic quantities.

Mineral objects are the building blocks for more complex objects in BurnMan. These objects are intended to represent minerals (or melts, or fluids) of fixed composition, with a well defined equation of state that defines the relationship between the current state (pressure and temperature) of the mineral and its thermodynamic potentials and derivatives (such as volume and entropy).

Mineral objects are initialized with a dictionary containing all of the parameters required by the desired equation of state. BurnMan contains implementations of may equations of state (*Equations of state*).

Uses:

- *Mineral databases*
- `burnman.Mineral`

Demonstrates:

- Different ways to define an endmember
- How to set state
- How to output thermodynamic and thermoelastic properties

5.1.6 example_calibrants

This example demonstrates the use of BurnMan's library of pressure calibrants. These calibrants are stripped-down versions of BurnMan's minerals, in that they are only designed to return pressure as a function of volume and temperature or volume as a function of pressure and temperature.

Uses:

- `burnman.classes.calibrant.Calibrant`
- `burnman.calibrants.tools.pressure_to_pressure()`
- Decker (1971) calibration for NaCl

Demonstrates:

- Use of the Calibrant class
- Conversion between pressures given by two different calibrations.

5.1.7 example_anisotropy

This example illustrates the basic functions required to convert an elastic stiffness tensor into elastic properties.

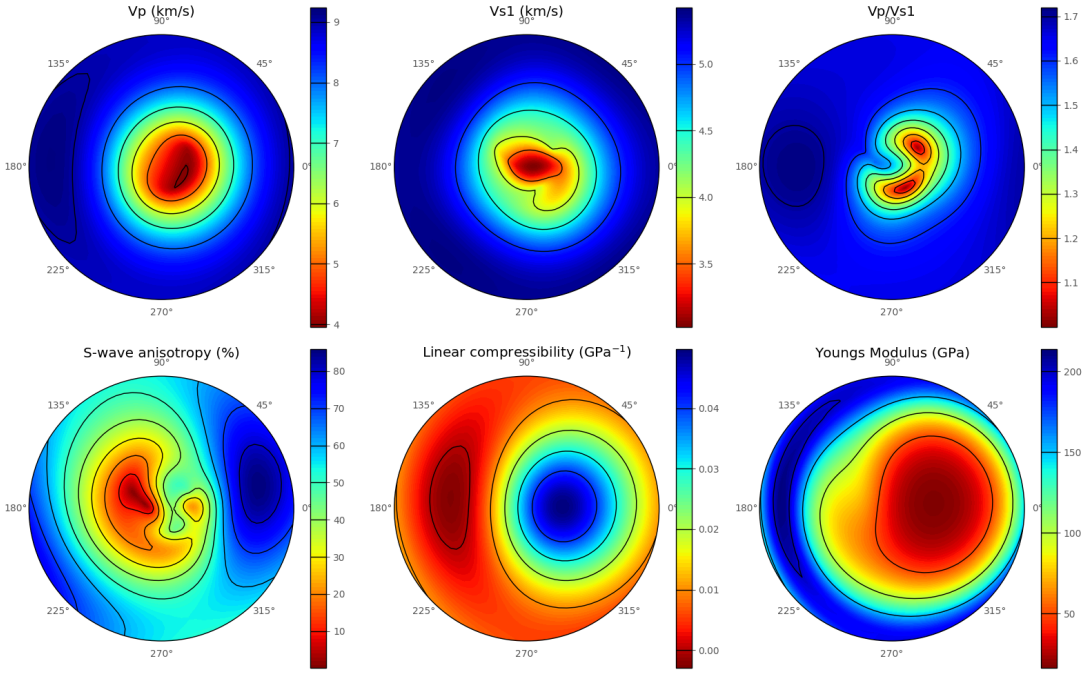
Specifically uses:

- *burnman.AnisotropicMaterial*

Demonstrates:

- anisotropic functions

Resulting figure:



5.1.8 example_anisotropic_mineral

This example illustrates how to create and interrogate an AnisotropicMineral object.

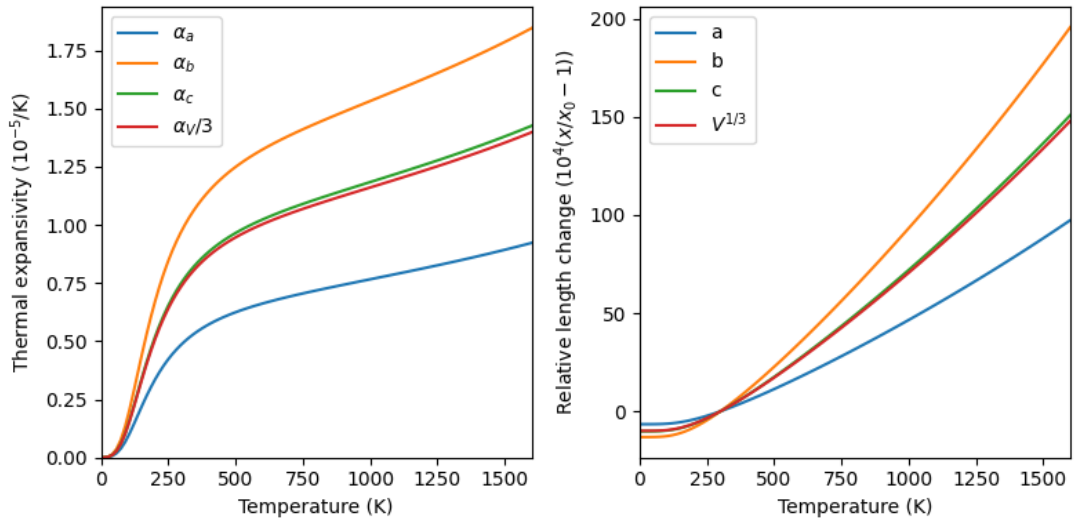
Specifically uses:

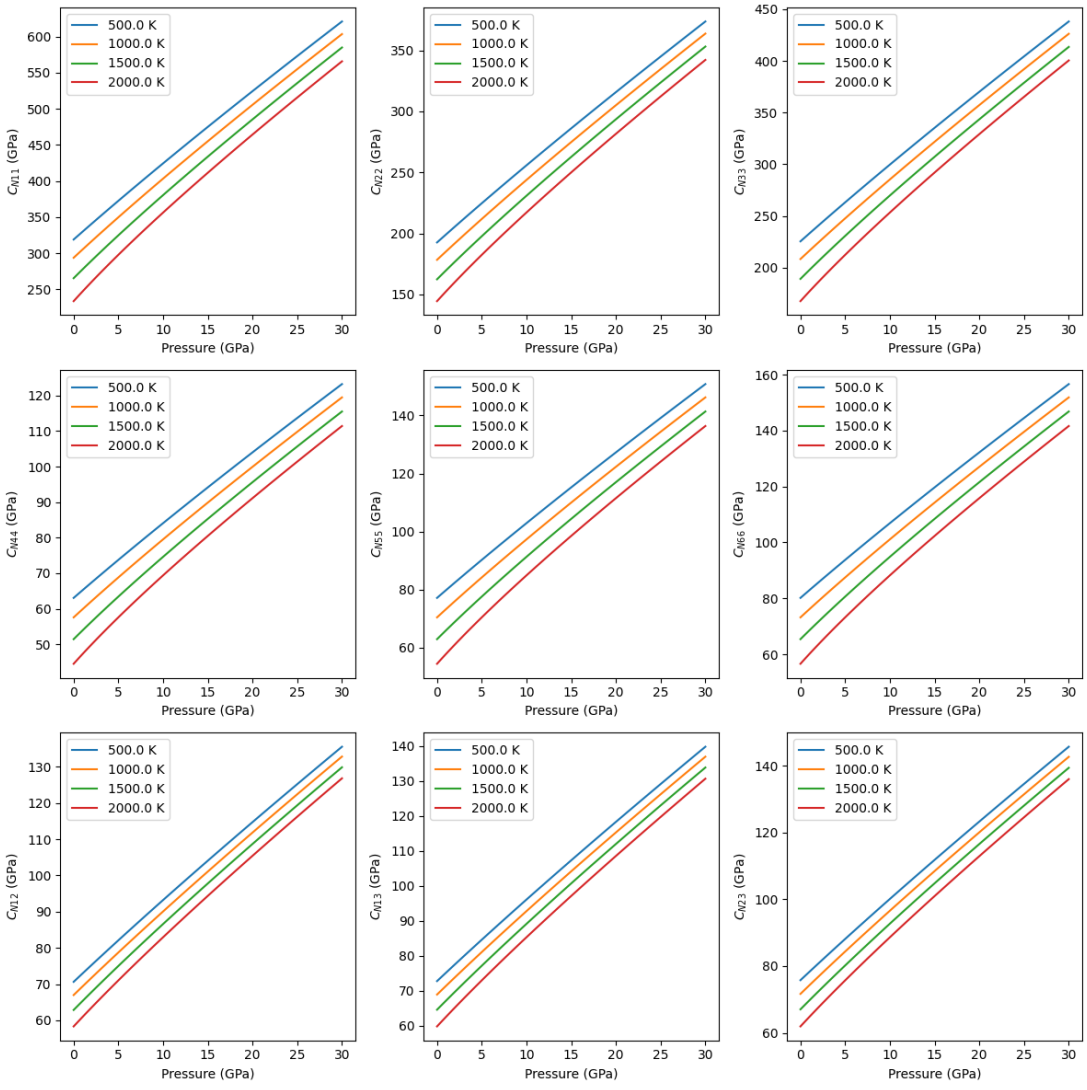
- *burnman.AnisotropicMineral*

Demonstrates:

- anisotropic functions

Resulting figure:


α_a
α_b
α_c
$\alpha_V/3$
Thermal expansivity (10^{-5}/K)
Temperature (K)
a
b
c
$V^{1/3}$
Relative length change ($10^4(x/x_0 - 1)$)
Temperature (K)


500.0 K
1000.0 K
1500.0 K
2000.0 K
C_{N11} (GPa)
C_{N22} (GPa)
C_{N33} (GPa)
C_{N44} (GPa)
C_{N55} (GPa)
C_{N66} (GPa)
C_{N12} (GPa)
C_{N13} (GPa)
C_{N23} (GPa)
Pressure (GPa)

5.1.9 example_geotherms

This example shows each of the geotherms currently possible with BurnMan. These are:

1. Brown and Shankland, 1981 [BS81]
2. Anderson, 1982 [And82]
3. Watson and Baxter, 2007 [WB07]
4. linear extrapolation
5. Read in from file from user
6. Adiabatic from potential temperature and choice of mineral

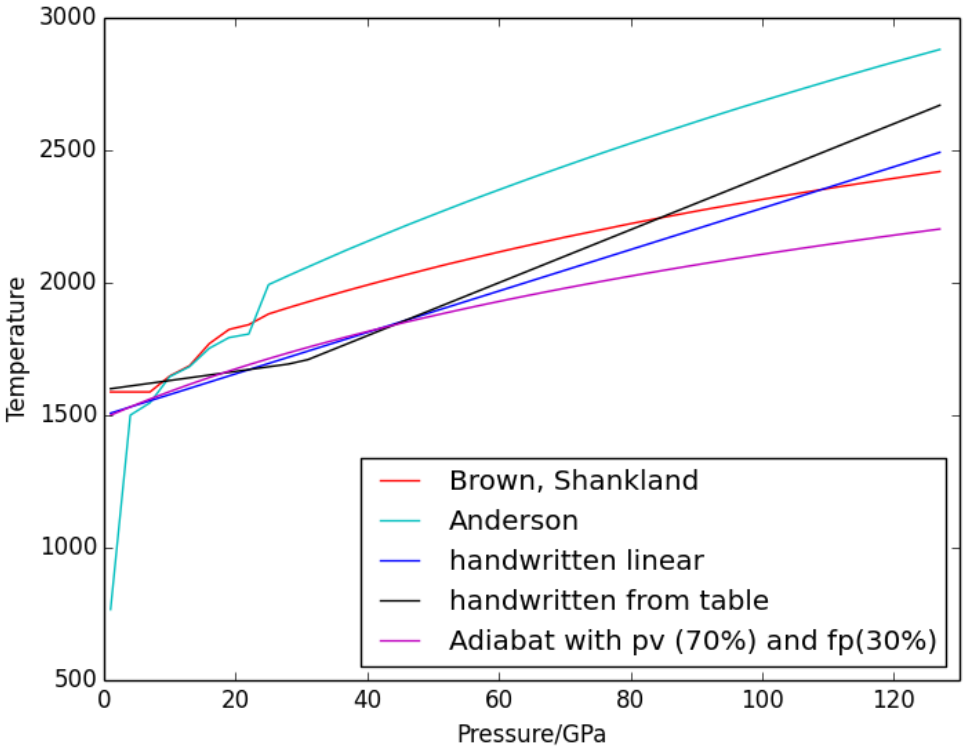
Uses:

- `burnman.geotherm.brown_shankland()`
- `burnman.geotherm.anderson()`
- input geotherm file *input_geotherm/example_geotherm.txt* (optional)
- *burnman.Composite* for adiabat

Demonstrates:

- the available geotherms

Resulting figure:



5.1.10 example_composition

This example script demonstrates the use of BurnMan's Composition class.

Uses:

- *burnman.Composition*

Demonstrates:

- Creating an instance of the Composition class with a molar or weight composition
- Printing weight, molar, atomic compositions
- Renormalizing compositions
- Modifying the independent set of components
- Modifying compositions by adding and removing components

5.2 Simple Examples

The following is a list of simple examples:

- *example_beginner*,
- *example_seismic*,
- *example_composite_seismic_velocities*,
- *example_averaging*, and
- *example_chemical_potentials*.

5.2.1 example_beginner

This example script is intended for absolute beginners to BurnMan. We cover importing BurnMan modules, creating a composite material, and calculating its seismic properties at lower mantle pressures and temperatures. Afterwards, we plot it against a 1D seismic model for visual comparison.

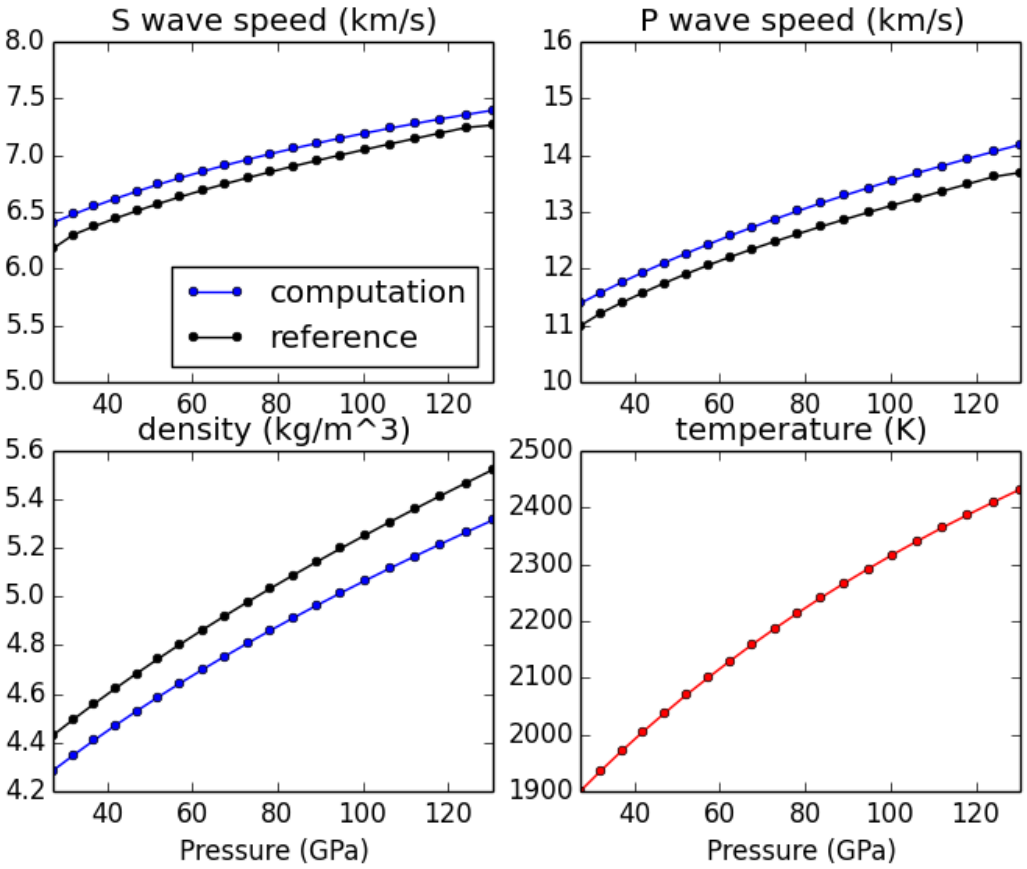
Uses:

- *Mineral databases*
- *burnman.Composite*
- `burnman.seismic.PREM`
- `burnman.geotherm.brown_shankland()`
- *burnman.Material.evaluate()*

Demonstrates:

- creating basic composites
- calculating thermoelastic properties
- seismic comparison

Resulting figure:



5.2.2 example_seismic

Shows the various ways to input seismic models (V_s, V_p, V_ϕ, ρ) as a function of depth (or pressure) as well as different velocity model libraries available within Burnman:

1. PREM [DA81]
2. STW105 [KED08]
3. AK135 [KEB95]
4. IASP91 [KE91]

This example will first calculate or read in a seismic model and plot the model along the defined pressure range. The example also illustrates how to import a seismic model of your choice, here shown by importing

AK135 [KEB95].

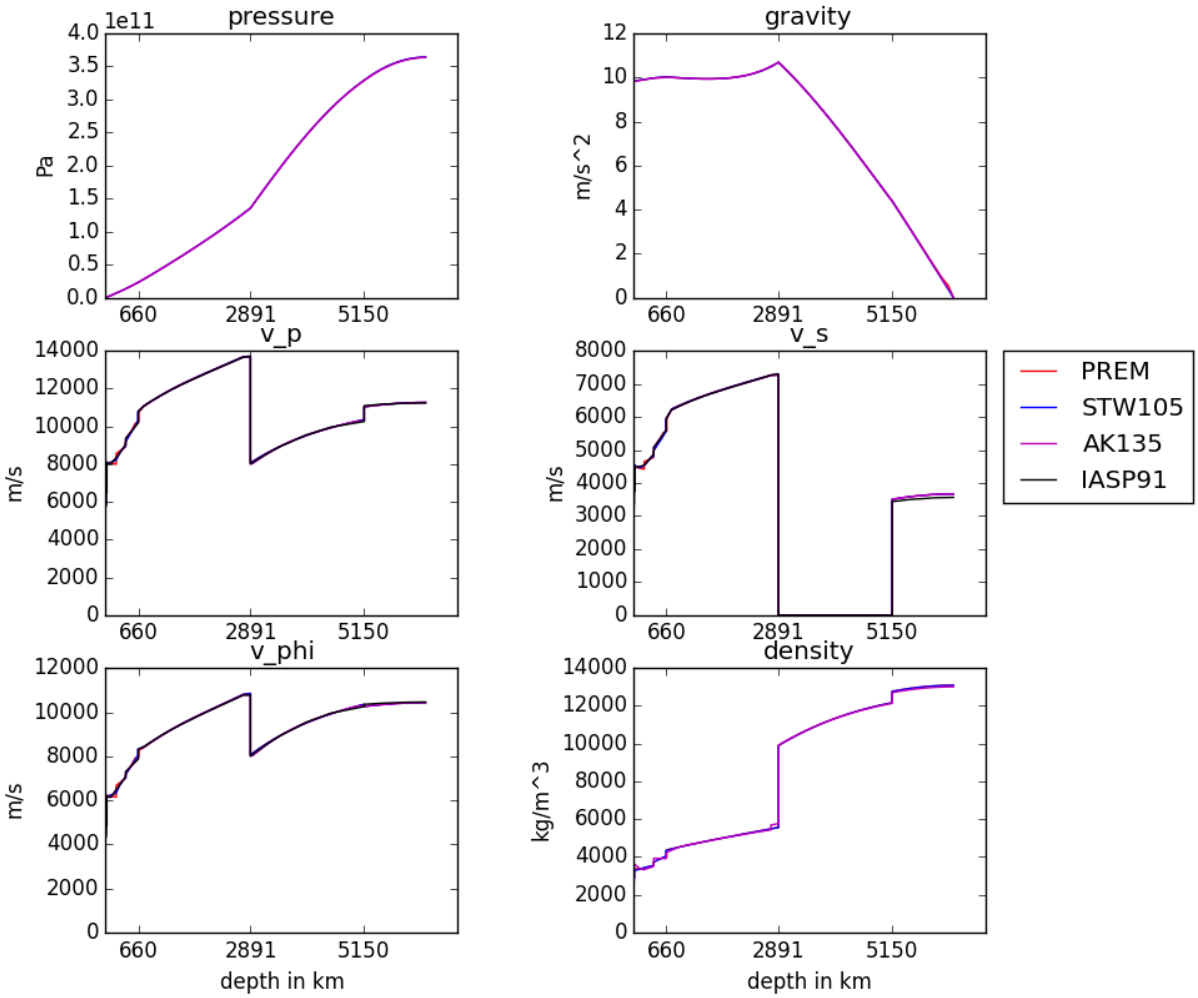
Uses:

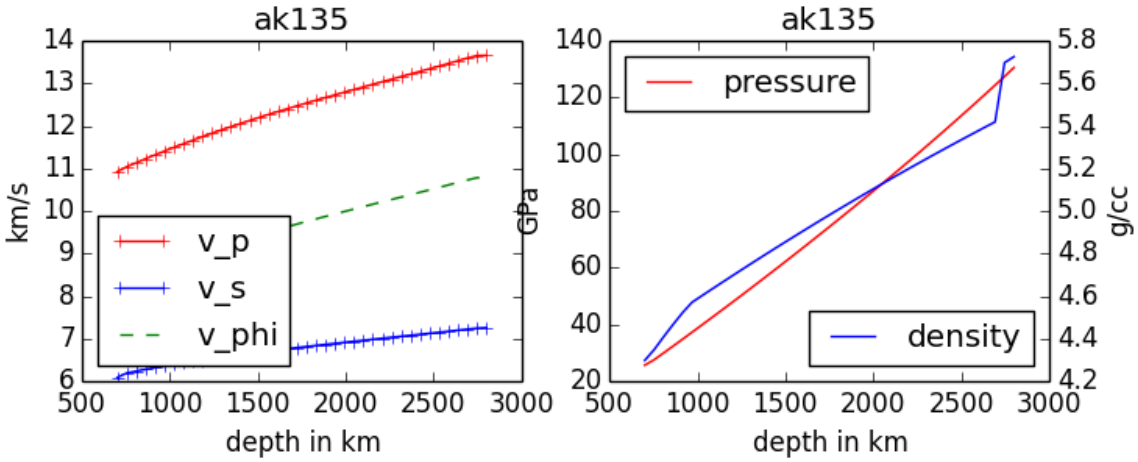
- *Seismic*

Demonstrates:

- Utilization of library seismic models within BurnMan
- Input of user-defined seismic models

Resulting figures:





5.2.3 example_composite_seismic_velocities

This example shows how to create different minerals, how to compute seismic velocities, and how to compare them to a seismic reference model.

There are many different ways in BurnMan to combine minerals into a composition. Here we present a couple of examples:

1. Two minerals mixed in simple mole fractions. Can be chosen from the BurnMan libraries or from user defined minerals (see example_user_input_material)
2. Example with three minerals
3. Using preset solutions
4. Defining your own solution

To turn a method of mineral creation "on" the first if statement above the method must be set to True, with all others set to False.

Note: These minerals can include a spin transition in (Mg,Fe)O, see example_spintransition.py for explanation of how to implement this

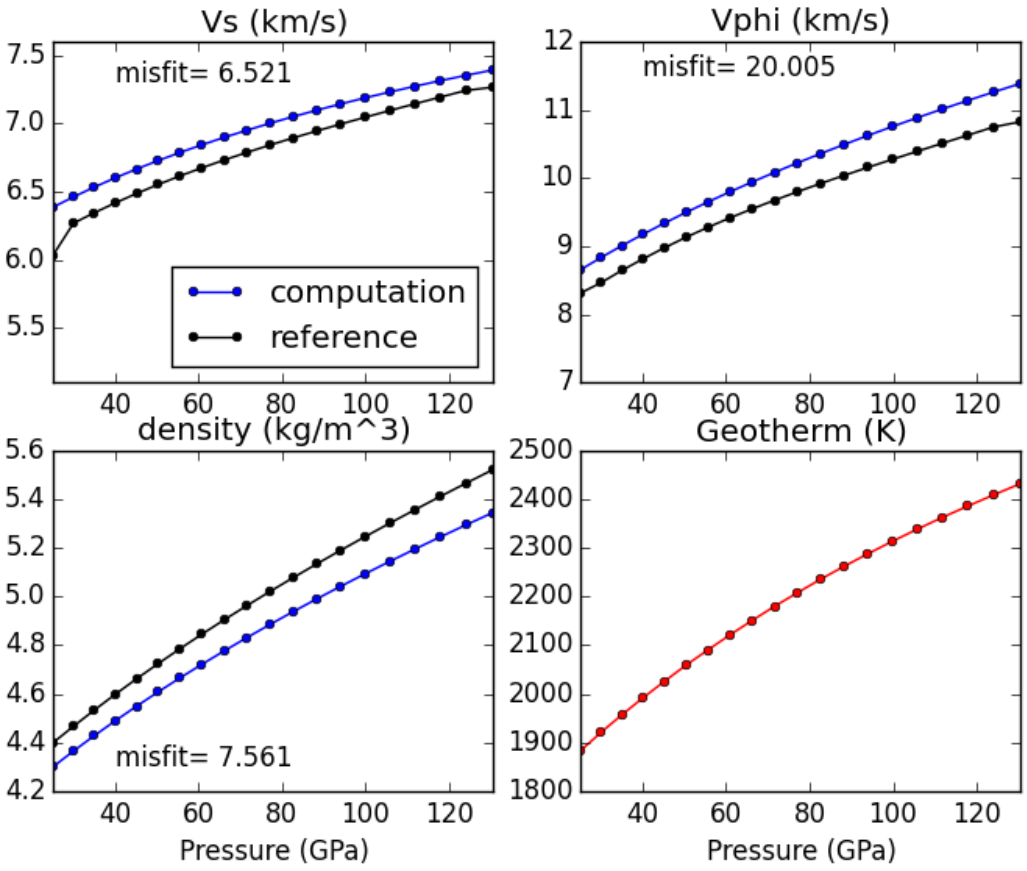
Uses:

- *Mineral databases*
- `burnman.Composite`
- `burnman.Mineral`
- `burnman.Solution`

Demonstrates:

- Different ways to define a composite
- Using minerals and solutions
- Compare computations to seismic models

Resulting figure:



5.2.4 example_averaging

This example shows the effect of different averaging schemes. Currently four averaging schemes are available:

1. Voight-Reuss-Hill
2. Voight averaging
3. Reuss averaging
4. Hashin-Shtrikman averaging

See [WDOConnell76] Journal of Geophysics and Space Physics for explanations of each averaging scheme.

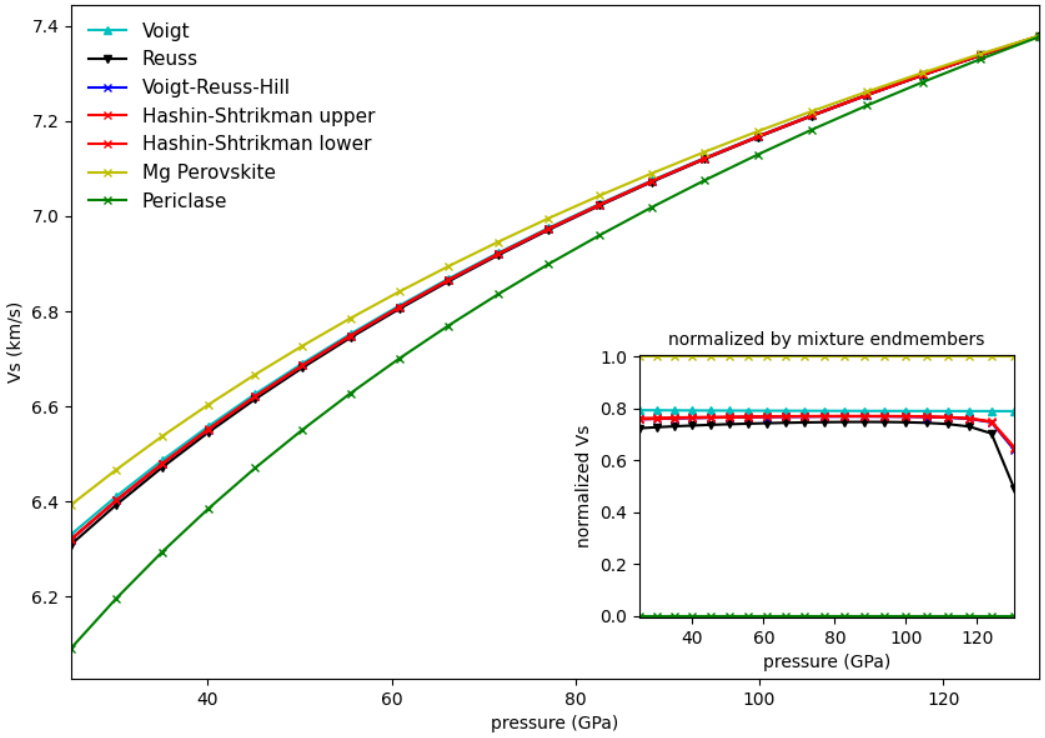
Specifically uses:

- *burnman.averaging_schemes.VoigtReussHill*
- *burnman.averaging_schemes.Voigt*
- *burnman.averaging_schemes.Reuss*
- *burnman.averaging_schemes.HashinShtrikmanUpper*
- *burnman.averaging_schemes.HashinShtrikmanLower*

Demonstrates:

- implemented averaging schemes

Resulting figure:



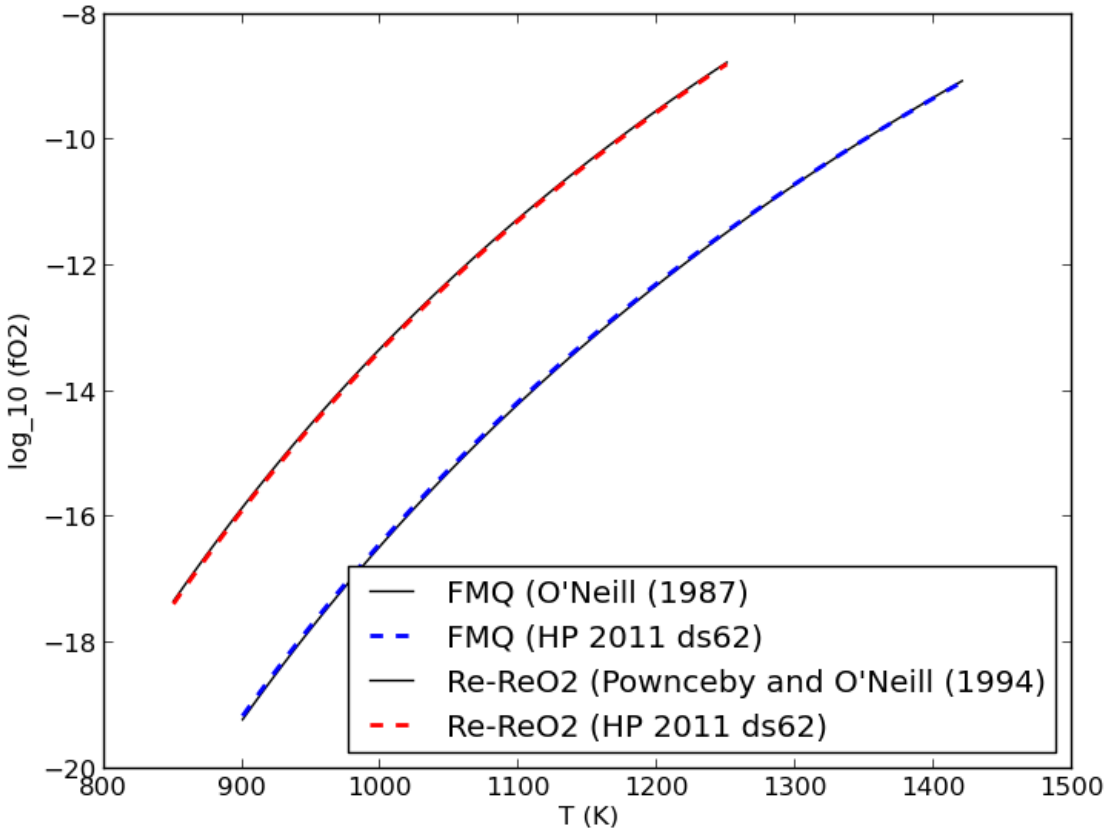
5.2.5 example_chemical_potentials

This example shows how to obtain chemical potentials and associated properties from an assemblage.

Demonstrates:

- How to calculate chemical potentials of an assemblage.
- How to compute fugacities and relative fugacities.

Resulting figure:



5.3 More Advanced Examples

Advanced examples:

- *example_spintransition*,
- *example_spintransition_thermal*,
- *example_user_input_material*,
- *example_optimize_pv*,
- *example_compare_all_methods*,
- *example_build_planet*,
- *example_fit_composition*,
- *example_fit_data*,
- *example_fit_eos*,
- *example_fit_solution*,

- *example_equilibrate*, and
- *example_olivine_binary*.

5.3.1 example_spintransition

This example shows the different minerals that are implemented with a spin transition. Minerals with spin transition are implemented by defining two separate minerals (one for the low and one for the high spin state). Then a third dynamic mineral is created that switches between the two previously defined minerals by comparing the current pressure to the transition pressure.

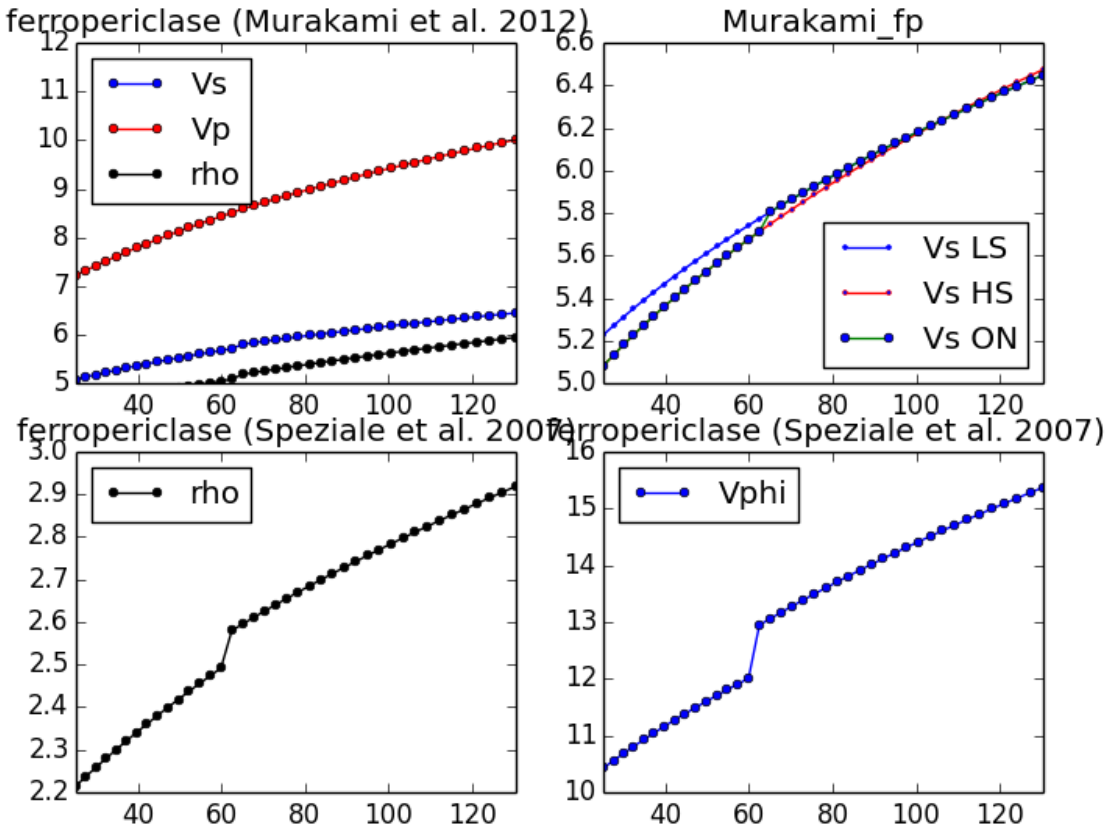
Specifically uses:

- `burnman.mineral_helpers.HelperSpinTransition()`
- *burnman.minerals.Murakami_etal_2012.fe_periclase()*
- *burnman.minerals.Murakami_etal_2012.fe_periclase_HS()*
- *burnman.minerals.Murakami_etal_2012.fe_periclase_LS()*

Demonstrates:

- implementation of spin transition in (Mg,Fe)O at user defined pressure

Resulting figure:



5.3.2 example_spintransition_thermal

This example illustrates how to create a non-ideal solution model for (Mg,Fe^{HS},Fe^{LS})O ferropericlase that has a gradual spin transition at finite temperature. First, we define the MgO endmember and two endmembers for the low and high spin states of FeO. Then we create a regular/symmetric solution that incorporates all three endmembers. The modified solution class contains a method called set_equilibrium_composition, which calculates the equilibrium proportions of the low and high spin phases at the desired bulk composition, pressure and temperature.

In this example, we neglect the elastic component of mixing. We also implicitly apply the Bragg-Williams approximation (i.e., we assume that there is no short-range order by only incorporating interactions that are a function of the average occupancy of species on each distinct site). Furthermore, the one site model [Mg,Fe^{HS},Fe^{LS}]O explicitly precludes long range order.

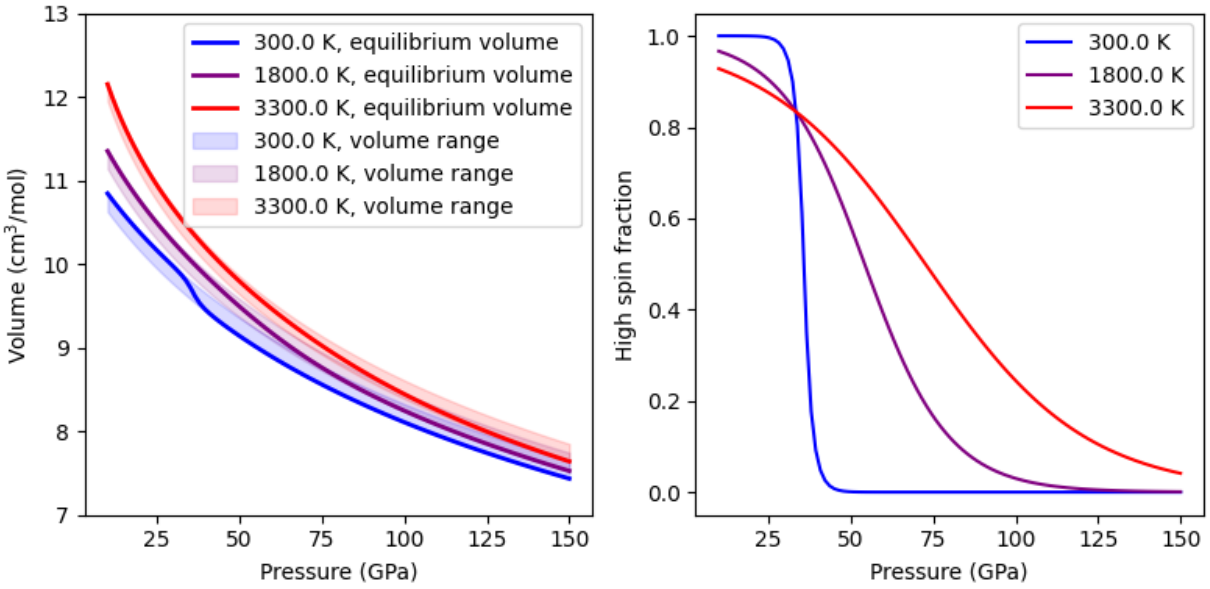
Specifically uses:

- *burnman.Mineral()*
- *burnman.Solution()*

Demonstrates:

- implementation of gradual spin transition in (Mg,Fe)O at a user-defined pressure and temperature

Resulting figure:



5.3.3 example_user_input_material

Shows user how to input a mineral of his/her choice without usint the library and which physical values need to be input for BurnMan to calculate V_P, V_Φ, V_S and density at depth.

Specifically uses:

- *burnman.Mineral*

Demonstrates:

- how to create your own minerals

5.3.4 example_optimize_pv

Vary the amount perovskite vs. ferropericlase and compute the error in the seismic data against PREM. For more extensive comments on this setup, see tutorial/step_2.py

Uses:

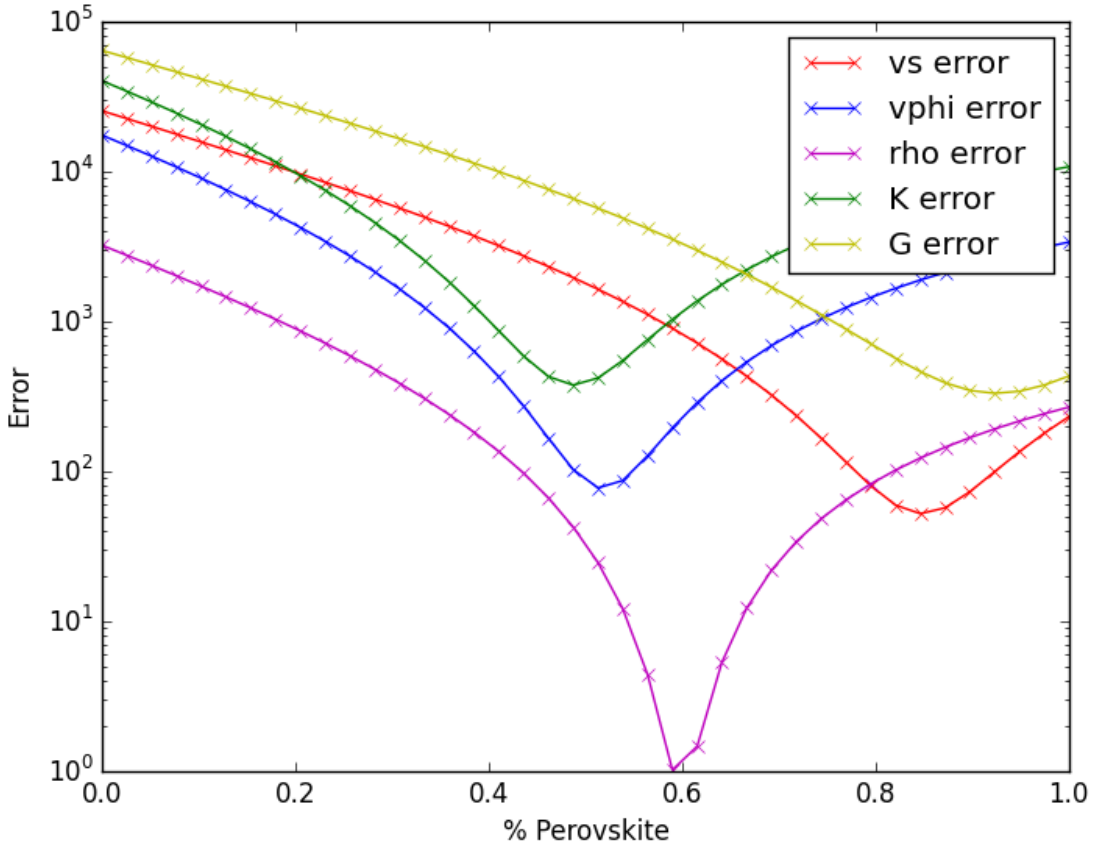
- *Mineral databases*
- *burnman.Composite*
- burnman.seismic.PREM
- burnman.geotherm.brown_shankland()
- *burnman.Material.evaluate()*

- *burnman.utils.math.compare_l2()*

Demonstrates:

- compare errors between models
- loops over models

Resulting figure:



5.3.5 example_compare_all_methods

This example demonstrates how to call each of the individual calculation methodologies that exist within BurnMan. See below for current options. This example calculates seismic velocity profiles for the same set of minerals and a plot of V_s, V_ϕ and ρ is produce for the user to compare each of the different methods.

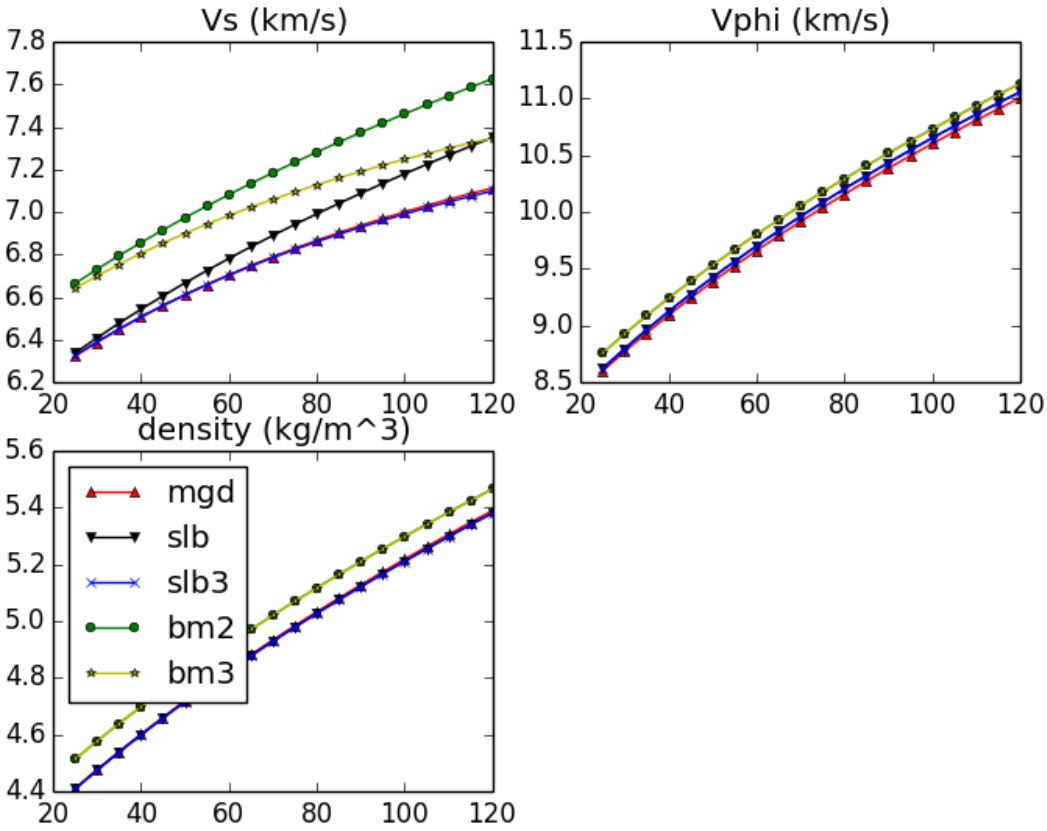
Specifically uses:

- *Equations of state*

Demonstrates:

- Each method for calculating velocity profiles currently included within BurnMan

Resulting figure:



5.3.6 example_build_planet

For Earth we have well-constrained one-dimensional density models. This allows us to calculate pressure as a function of depth. Furthermore, petrologic data and assumptions regarding the convective state of the planet allow us to estimate the temperature.

For planets other than Earth we have much less information, and in particular we know almost nothing about the pressure and temperature in the interior. Instead, we tend to have measurements of things like mass, radius, and moment-of-inertia. We would like to be able to make a model of the planet's interior that is consistent with those measurements.

However, there is a difficulty with this. In order to know the density of the planetary material, we need to know the pressure and temperature. In order to know the pressure, we need to know the gravity profile. And in order to the the gravity profile, we need to know the density. This is a nonlinear problem which requires us to iterate to find a self-consistent solution.

This example allows the user to define layers of planets of known outer radius and self- consistently solve for the density, pressure and gravity profiles. The calculation will iterate until the difference between central pressure calculations are less than 1e-5. The planet class in BurnMan (../burnman/planet.py) allows users to

call multiple properties of the model planet after calculations, such as the mass of an individual layer, the total mass of the planet and the moment if inertia. See planets.py for information on each of the parameters which can be called.

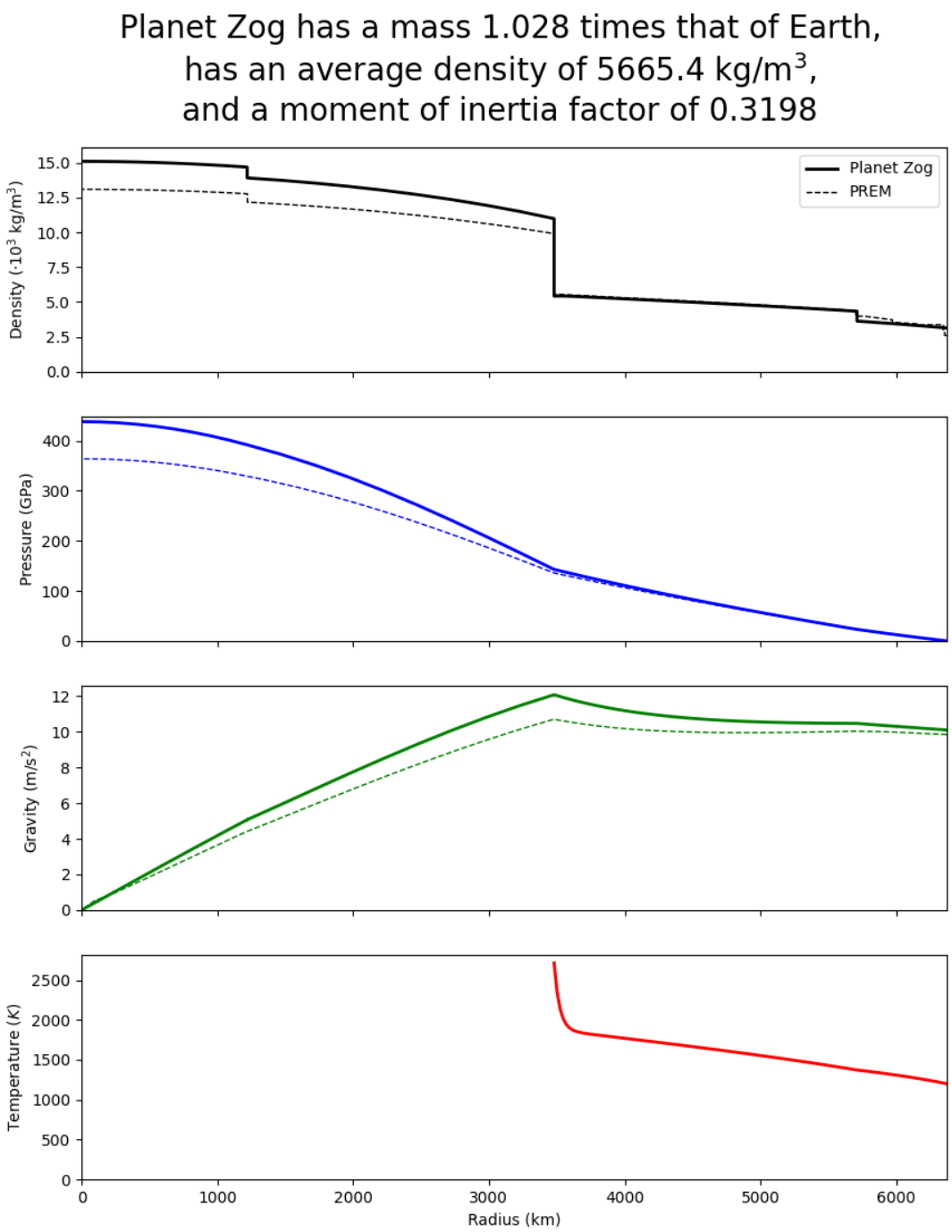
Uses:

- *Mineral databases*
- `burnman.Planet`
- `burnman.Layer`

Demonstrates:

- setting up a planet
- computing its self-consistent state
- computing various parameters for the planet
- seismic comparison

Resulting figure:


Planet Zog has a mass 1.028 times that of Earth,
has an average density of 5665.4 kg/m^3,
and a moment of inertia factor of 0.3198
Planet Zog
PREM
Density ($\cdot 10^3$ kg/m^3)
Pressure (GPa)
Gravity (m/s^2)
Temperature (K)
Radius (km)

5.3.7 example_fit_composition

This example shows how to fit compositional data to a solution model, how to partition a bulk composition between phases of known composition, and how to assess goodness of fit.

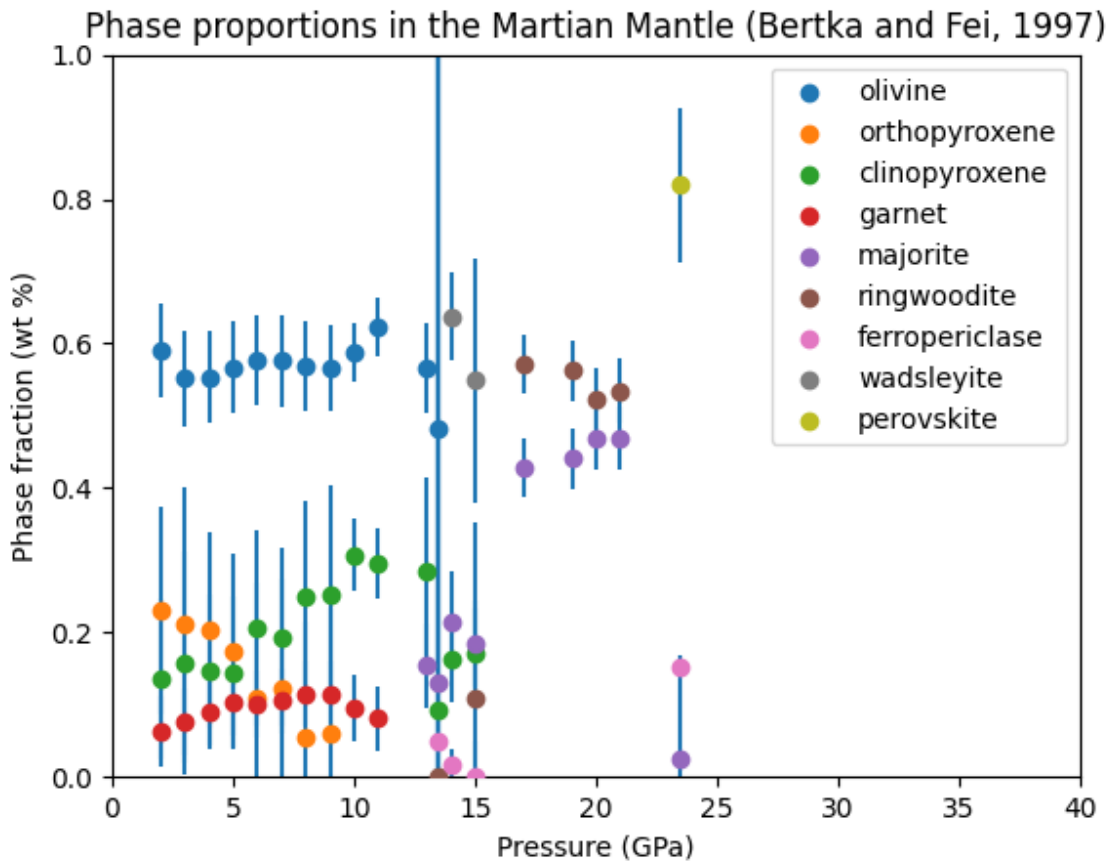
Uses:

- *burnman.Composition*
- *burnman.Solution*
- *burnman.optimize.composition_fitting.fit_phase_proportions_to_bulk_composition()*
- *burnman.optimize.composition_fitting.fit_composition_to_solution()*

Demonstrates:

- Fitting compositional data to a solution model
- Partitioning of a bulk composition between phases of known composition
- Assessing goodness of fit.

Resulting figure:



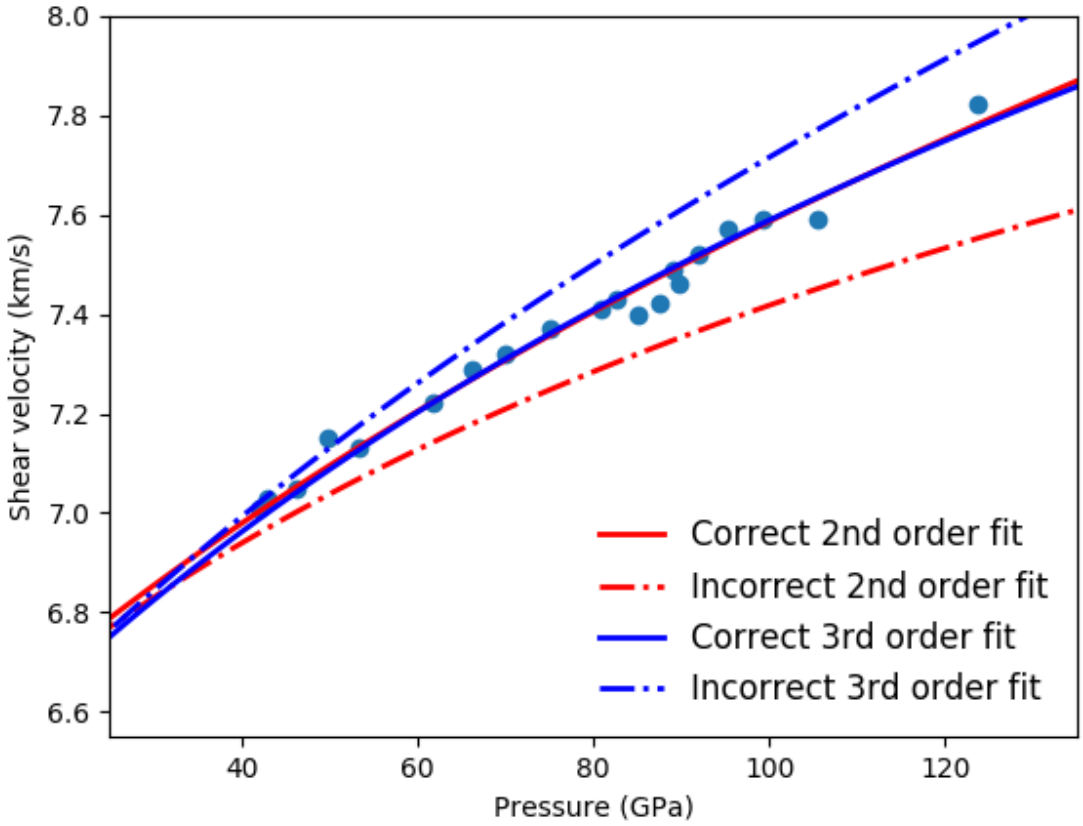
5.3.8 example_fit_data

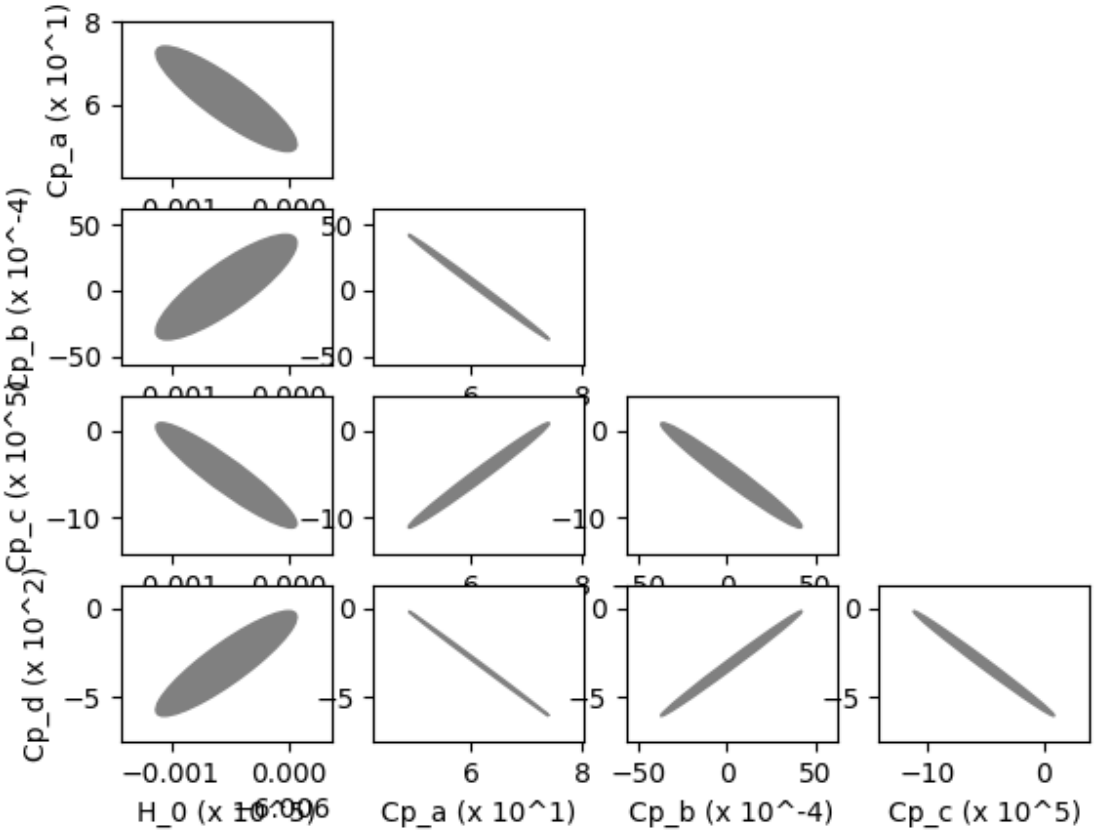
This example demonstrates BurnMan's functionality to fit various mineral physics data to an EoS of the user's choice.

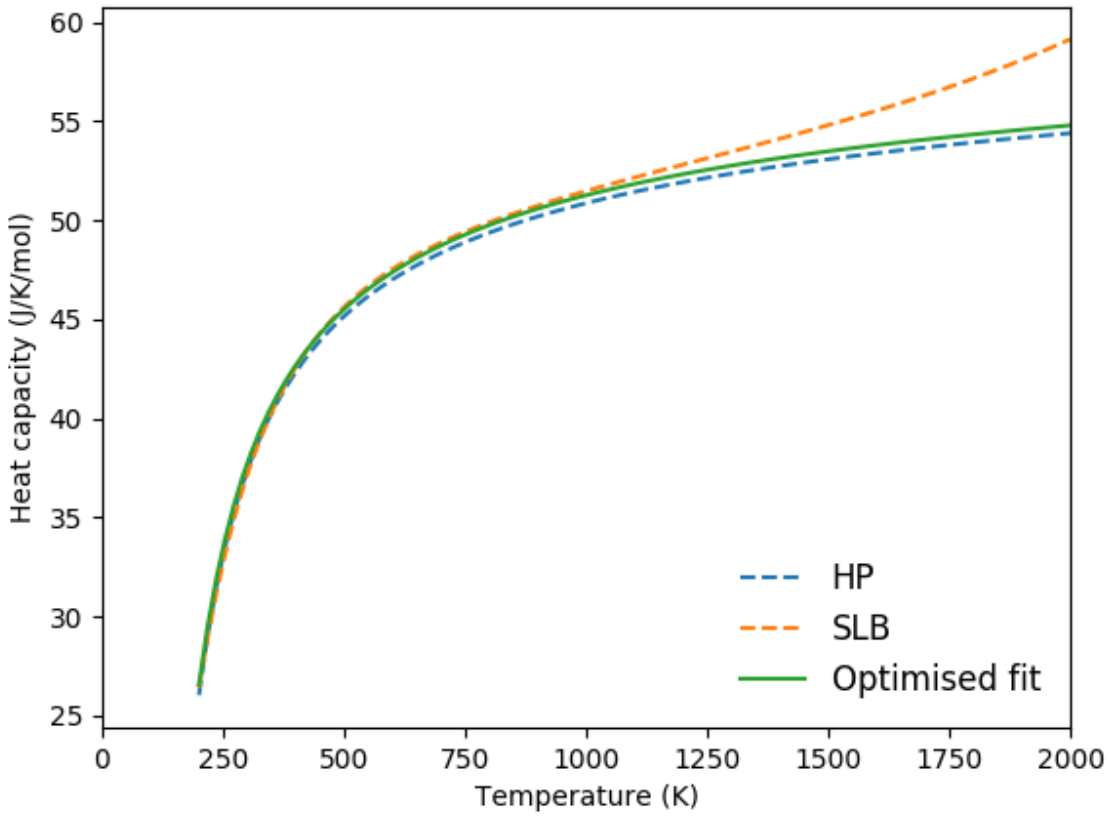
Please note also the separate file example_fit_eos.py, which can be viewed as a more advanced example in the same general field.

teaches: - least squares fitting

Resulting figures:




Cp_a (x 10^1)
Cp_b (x 10^-4)
Cp_c (x 10^5)
Cp_d (x 10^2)
H_0 (x 10^-5)
Cp_a (x 10^1)
Cp_b (x 10^-4)
Cp_c (x 10^5)



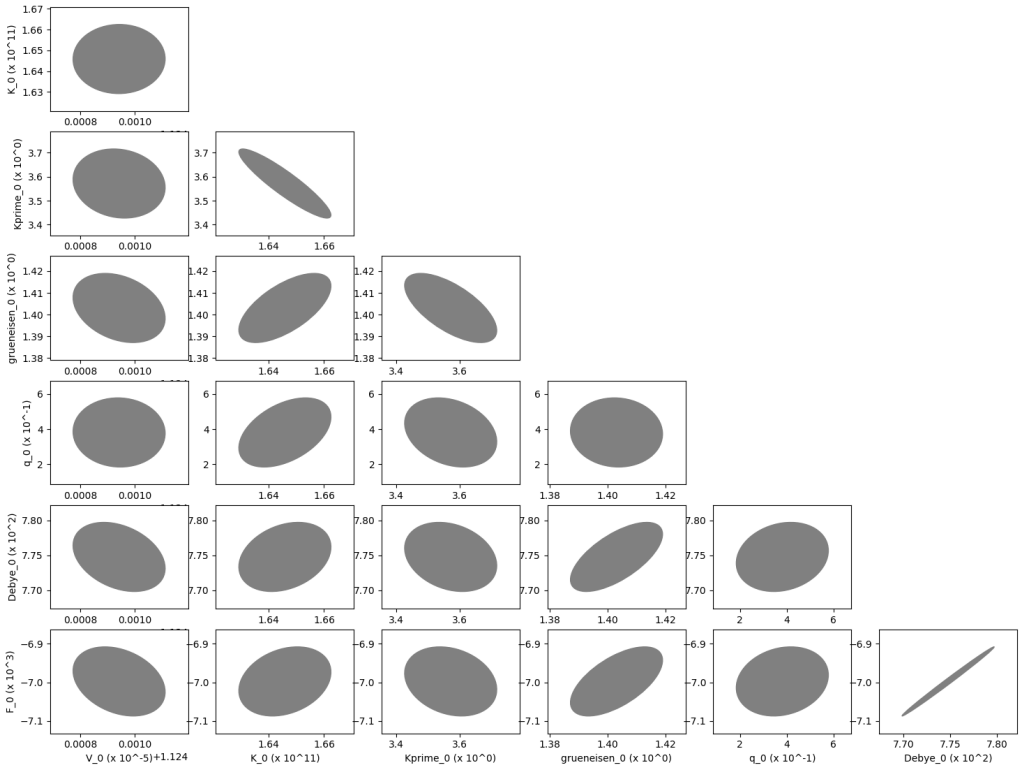
5.3.9 example_fit_eos

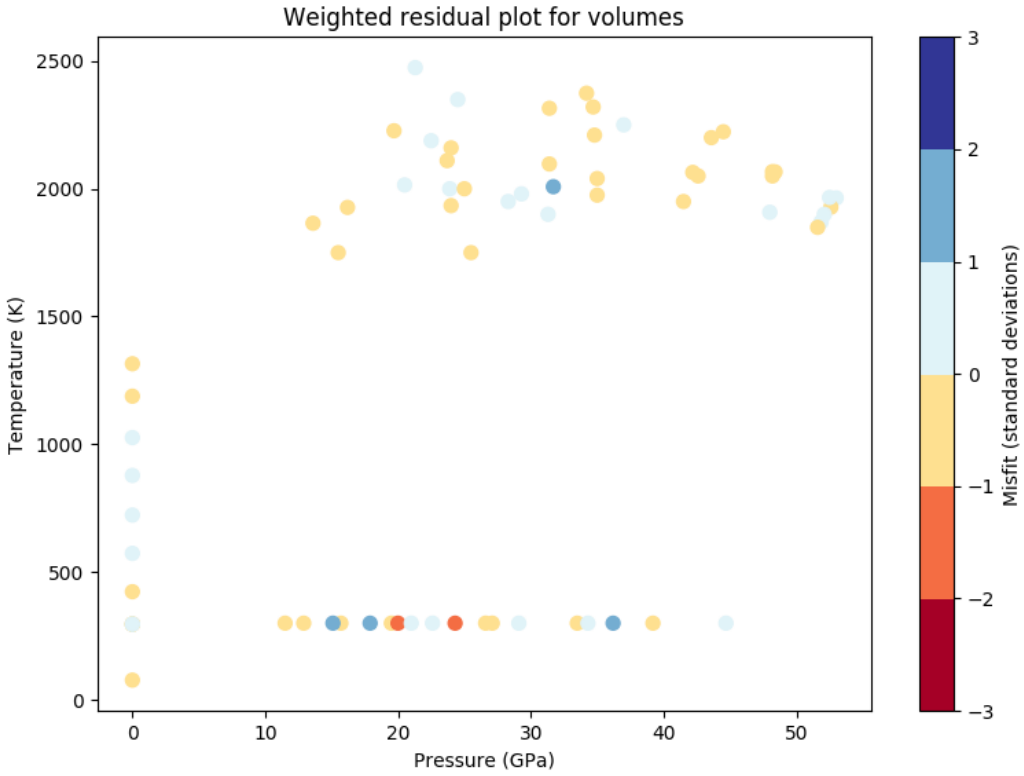
This example demonstrates BurnMan's functionality to fit data to an EoS of the user's choice.

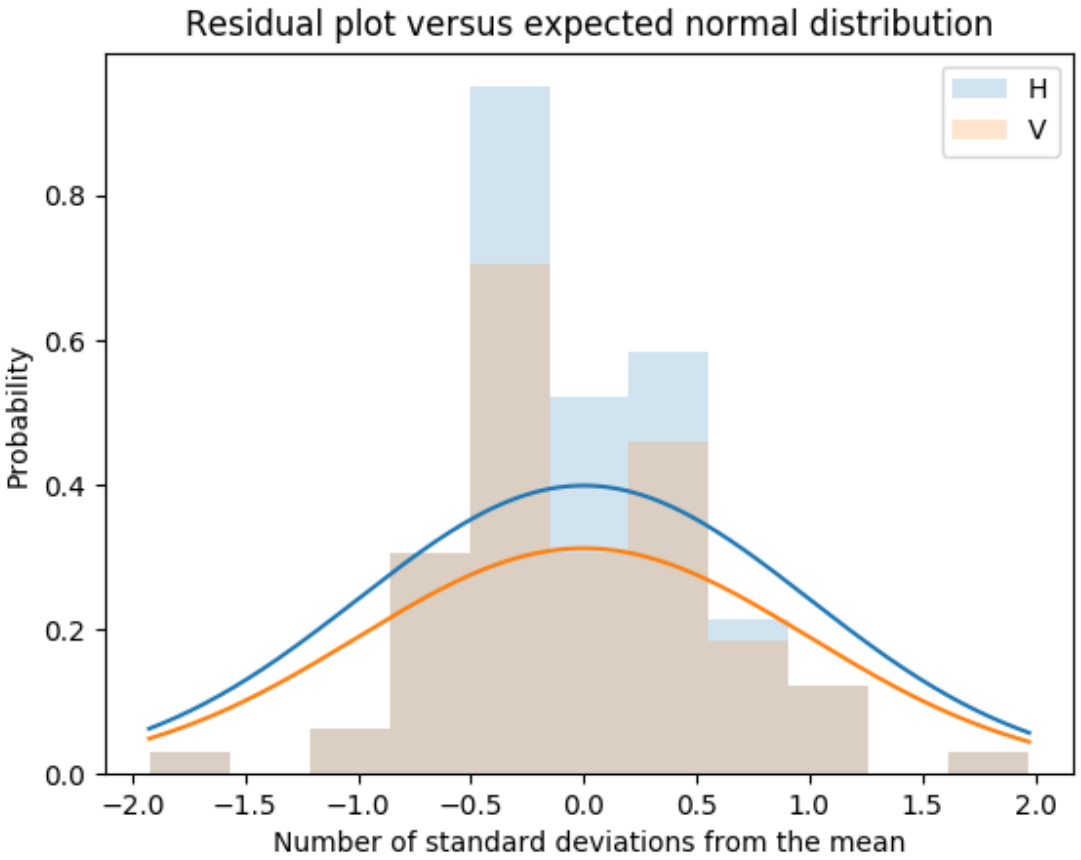
The first example deals with simple PVT fitting. The second example illustrates how powerful it can be to provide non-PVT constraints to the same fitting problem.

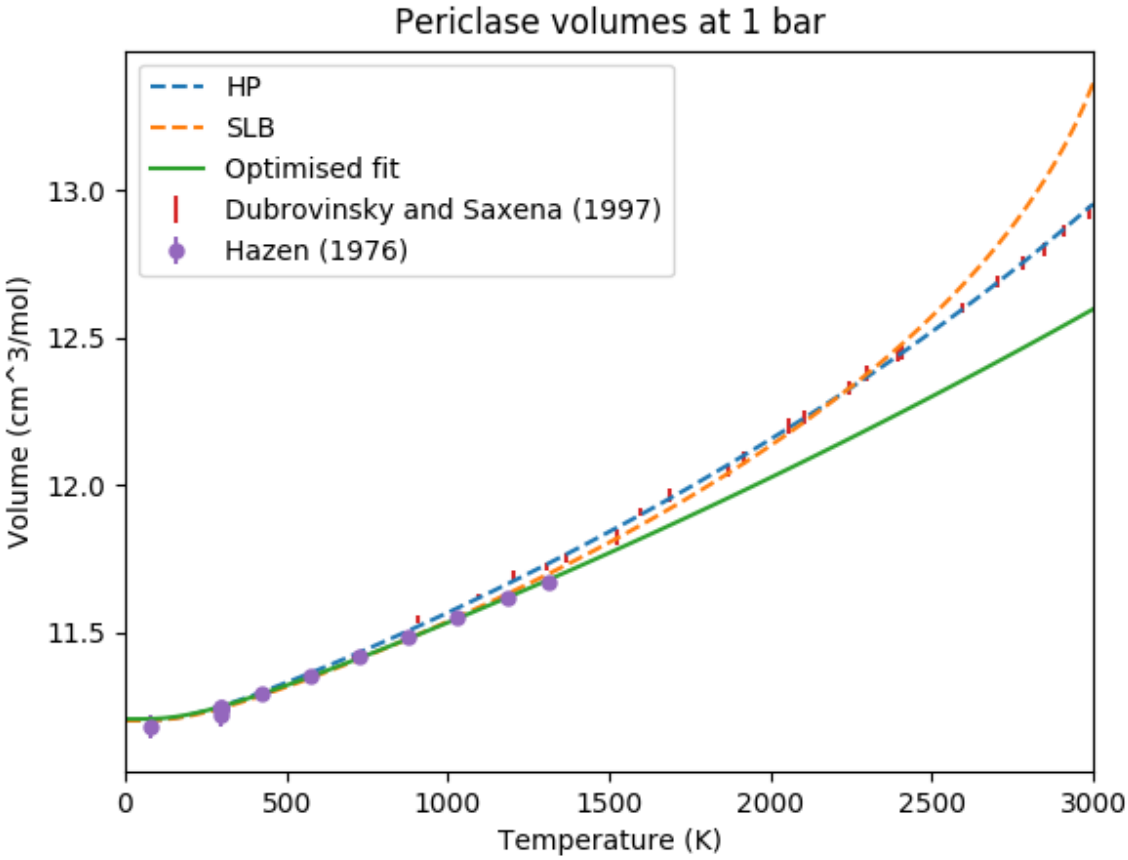
teaches: - least squares fitting

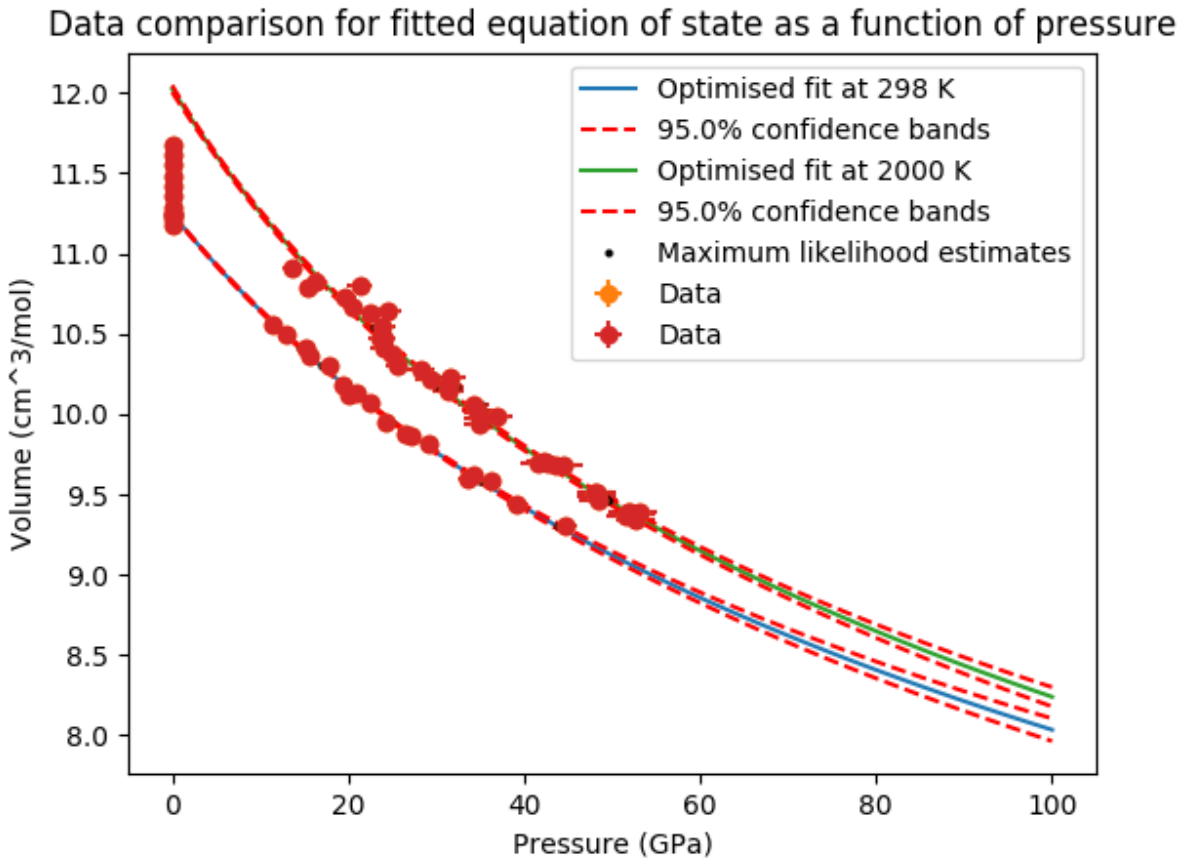
Last seven resulting figures:

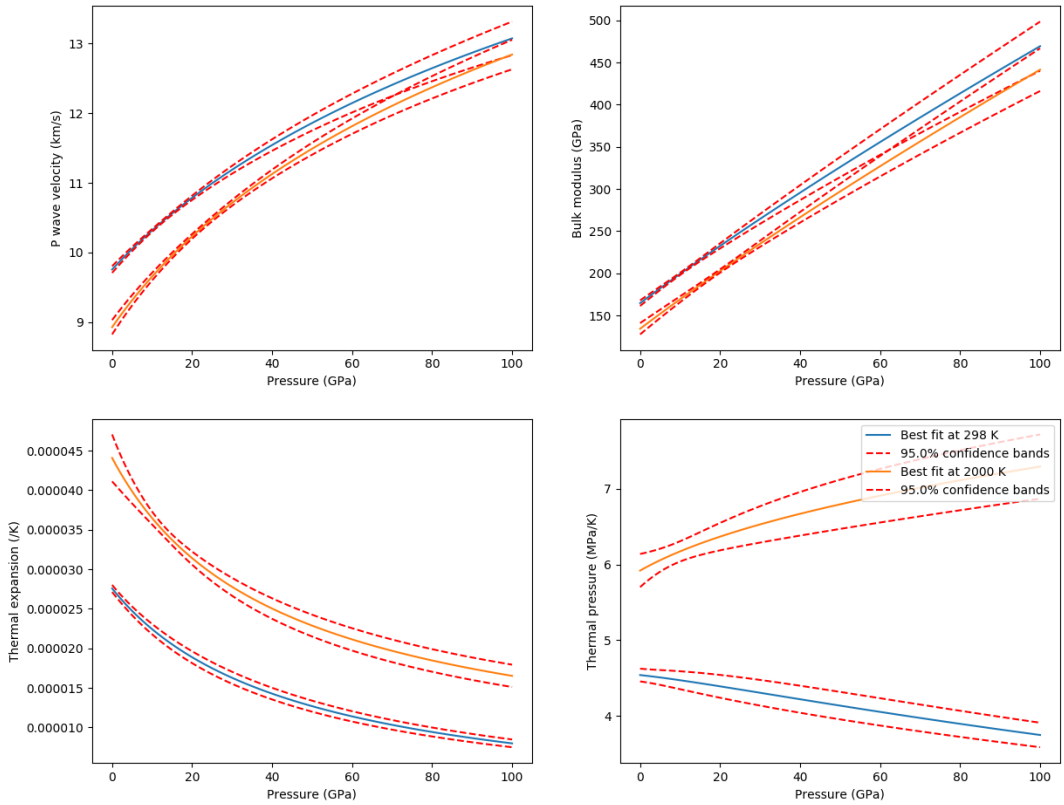

K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)
F_0 (x 10^3)
V_0 (x 10^-5)+1.124
K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)

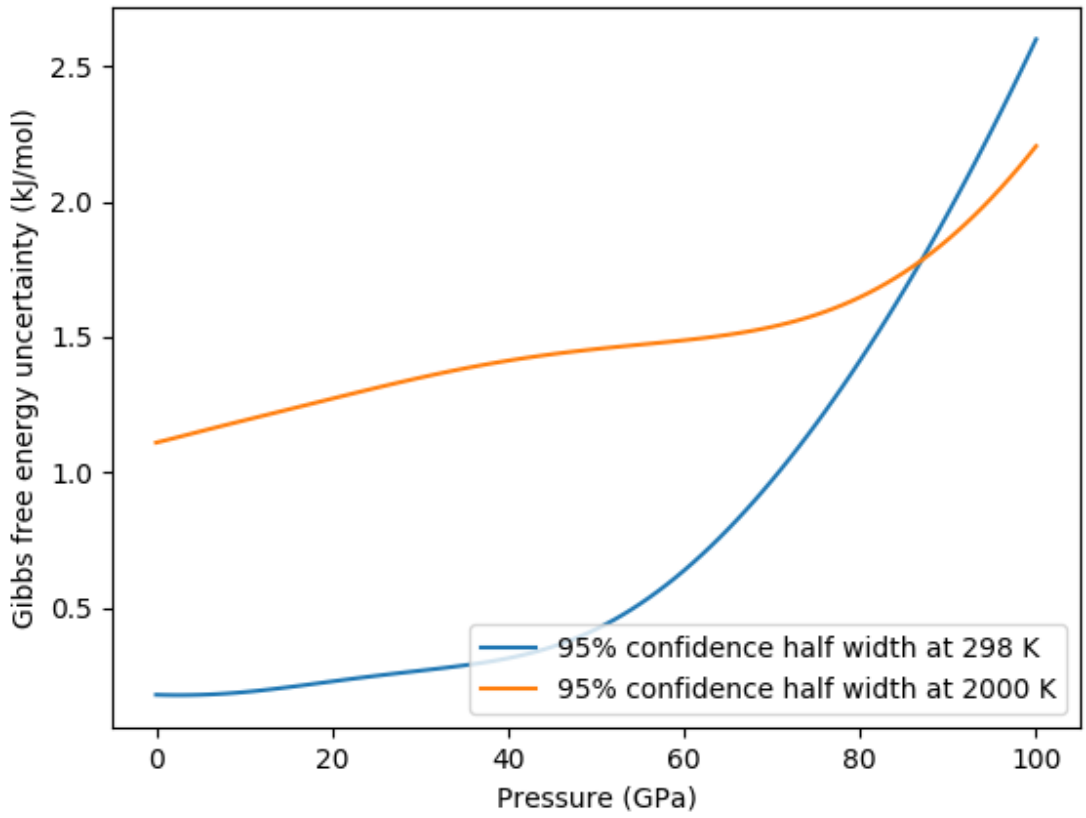

Weighted residual plot for volumes
Temperature (K)
Pressure (GPa)
Misfit (standard deviations)
2500
2000
1500
1000
500
0
0
10
20
30
40
50
3
2
1
0
−1
−2
−3


Residual plot versus expected normal distribution
H
V
Probability
0.0
0.2
0.4
0.6
0.8
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
Number of standard deviations from the mean


Periclase volumes at 1 bar
HP
SLB
Optimised fit
Dubrovinsky and Saxena (1997)
Hazen (1976)
Volume (cm^3/mol)
13.0
12.5
12.0
11.5
0
500
1000
1500
2000
2500
3000
Temperature (K)


Data comparison for fitted equation of state as a function of pressure
Optimised fit at 298 K
95.0% confidence bands
Optimised fit at 2000 K
95.0% confidence bands
Maximum likelihood estimates
Data
Data
Volume (cm^3/mol)
Pressure (GPa)
12.0
11.5
11.0
10.5
10.0
9.5
9.0
8.5
8.0
0
20
40
60
80
100


P wave velocity (km/s)
Pressure (GPa)
Bulk modulus (GPa)
Pressure (GPa)
Thermal expansion (/K)
Pressure (GPa)
Thermal pressure (MPa/K)
Pressure (GPa)
Best fit at 298 K
95.0% confidence bands
Best fit at 2000 K
95.0% confidence bands



5.3.10 example_fit_solution

This example demonstrates how to fit parameters for solution models using a range of compositionally-variable experimental data.

The example in this file deals with finding optimized parameters for the forsterite-fayalite binary using a mixture of volume and seismic velocity data.

teaches: - least squares fitting for solution data

Resulting figures:

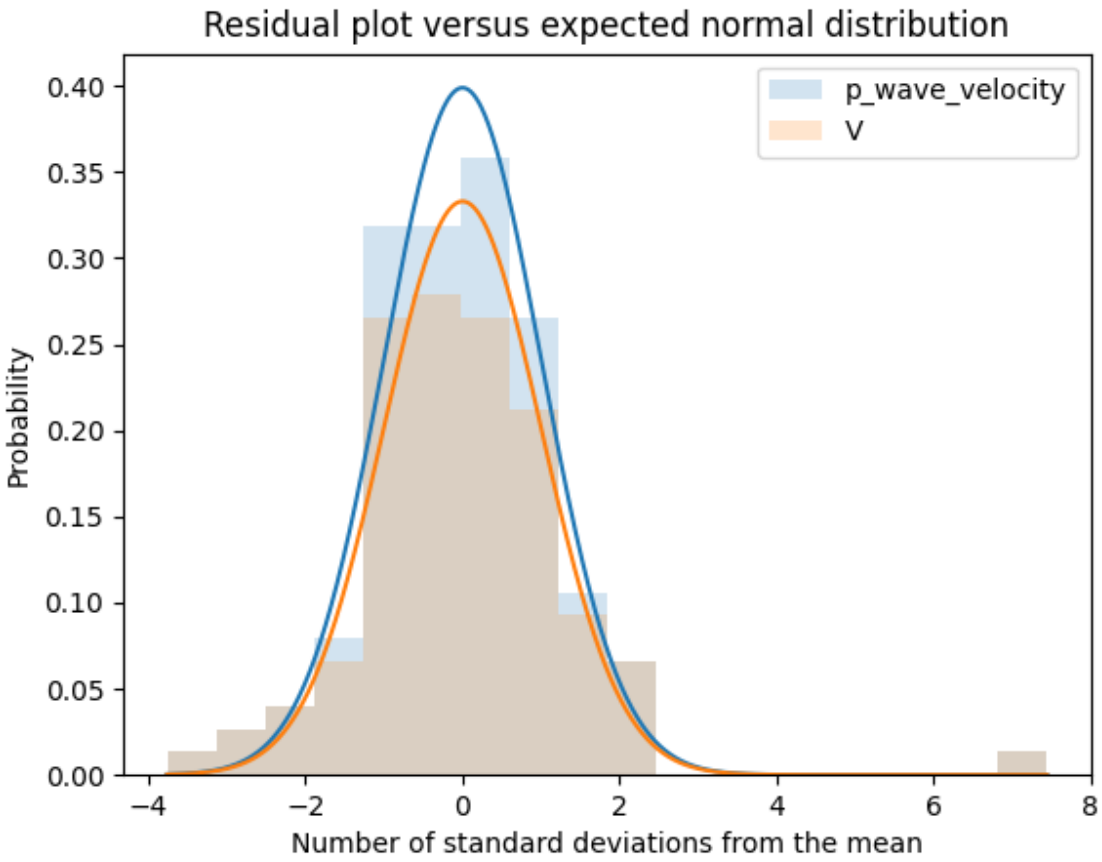


Fig. 1: Data residuals relative to the model fitted using all the provided data.

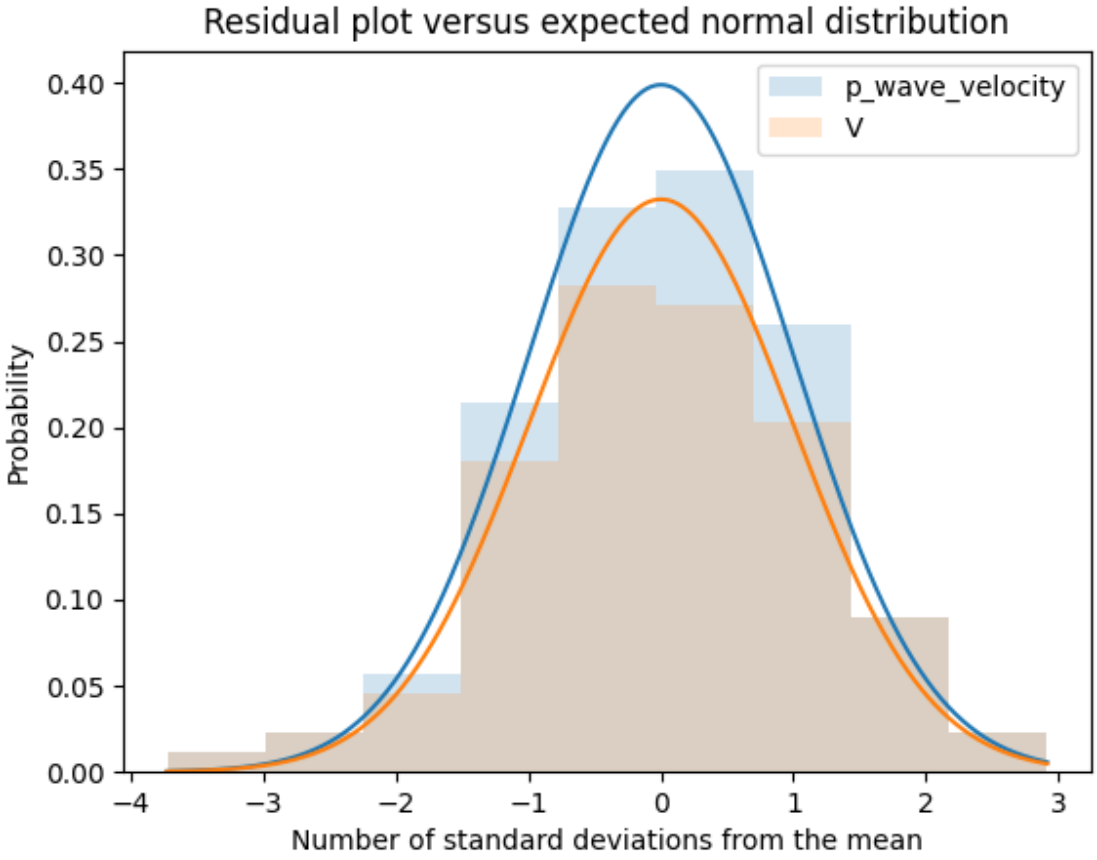


Fig. 2: Data residuals relative to the model fitted using the provided data after semi-automatic removal of spurious data.

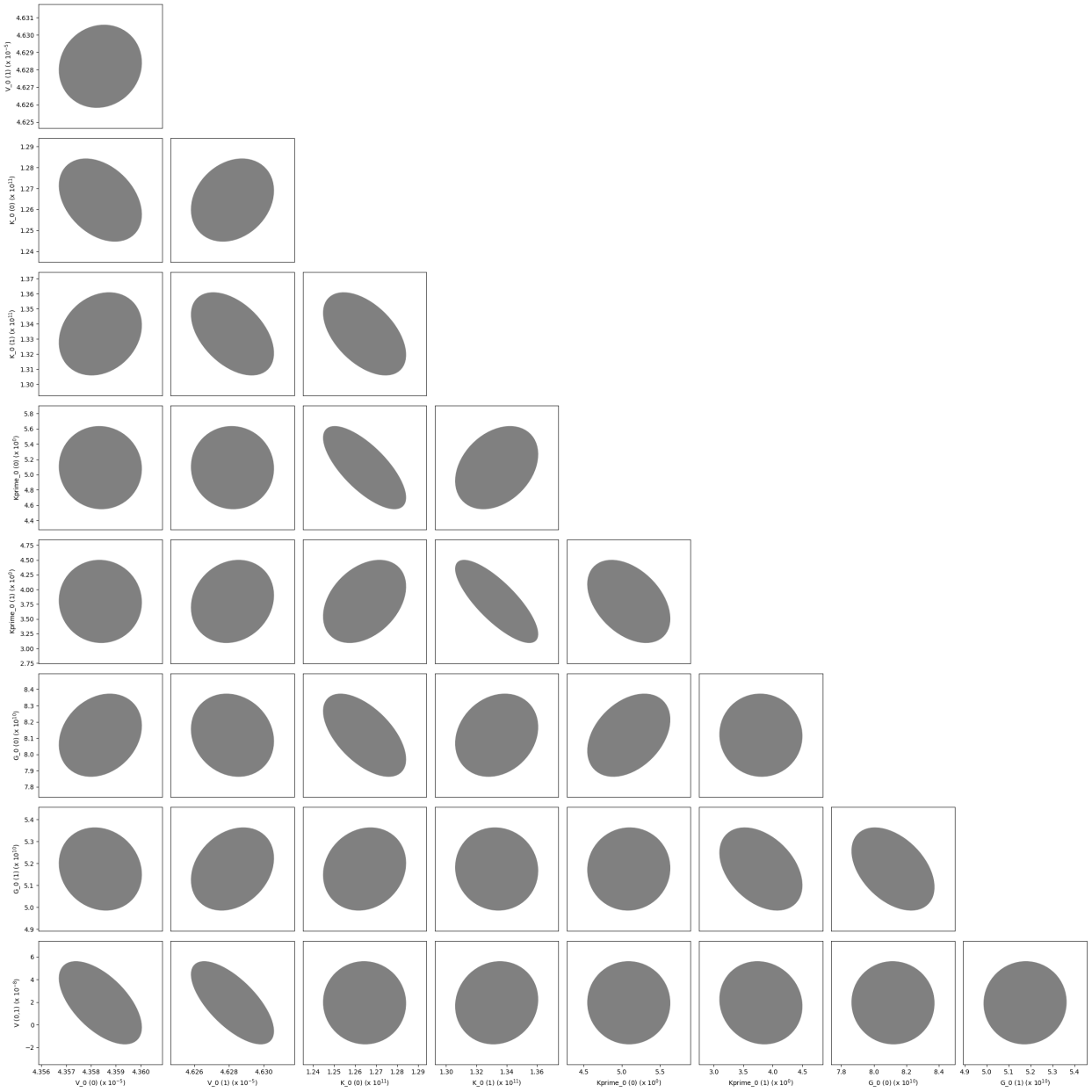


Fig. 3: The variance-covariance matrix of the optimized parameters shown as a corner plot.

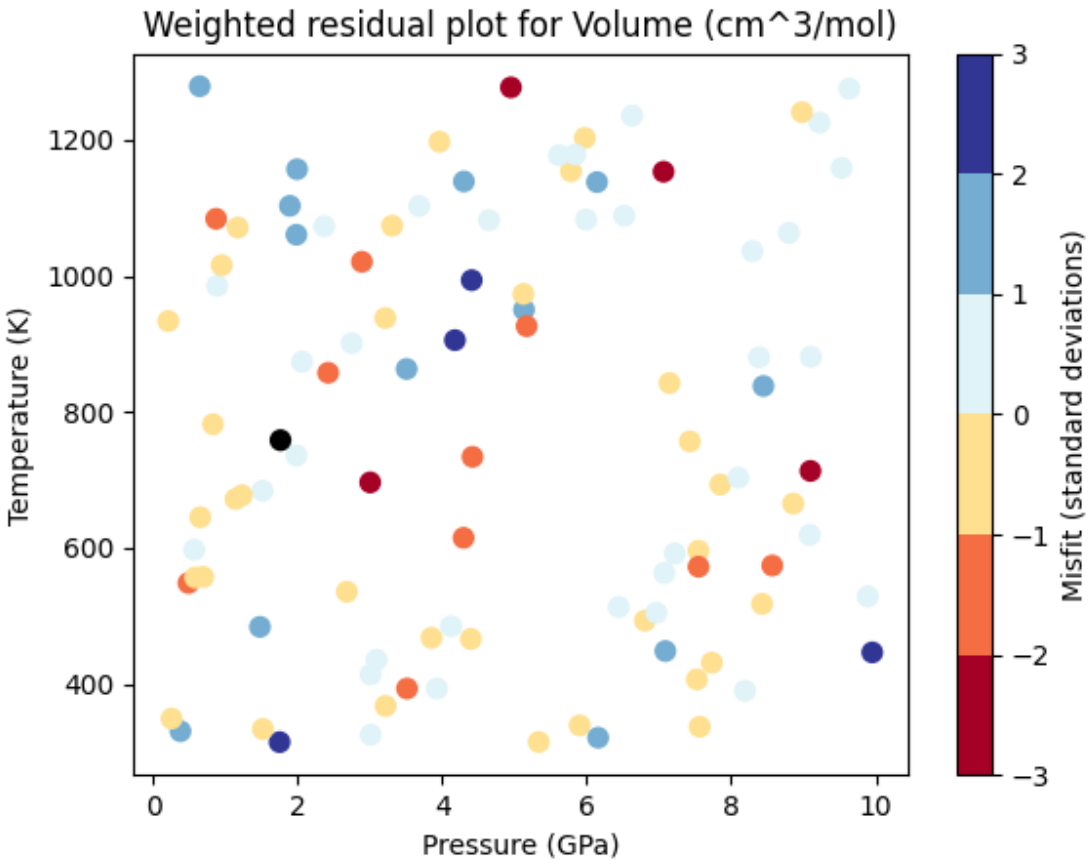


Fig. 4: A P-T plot showing the weighted residuals of each piece of volume data.

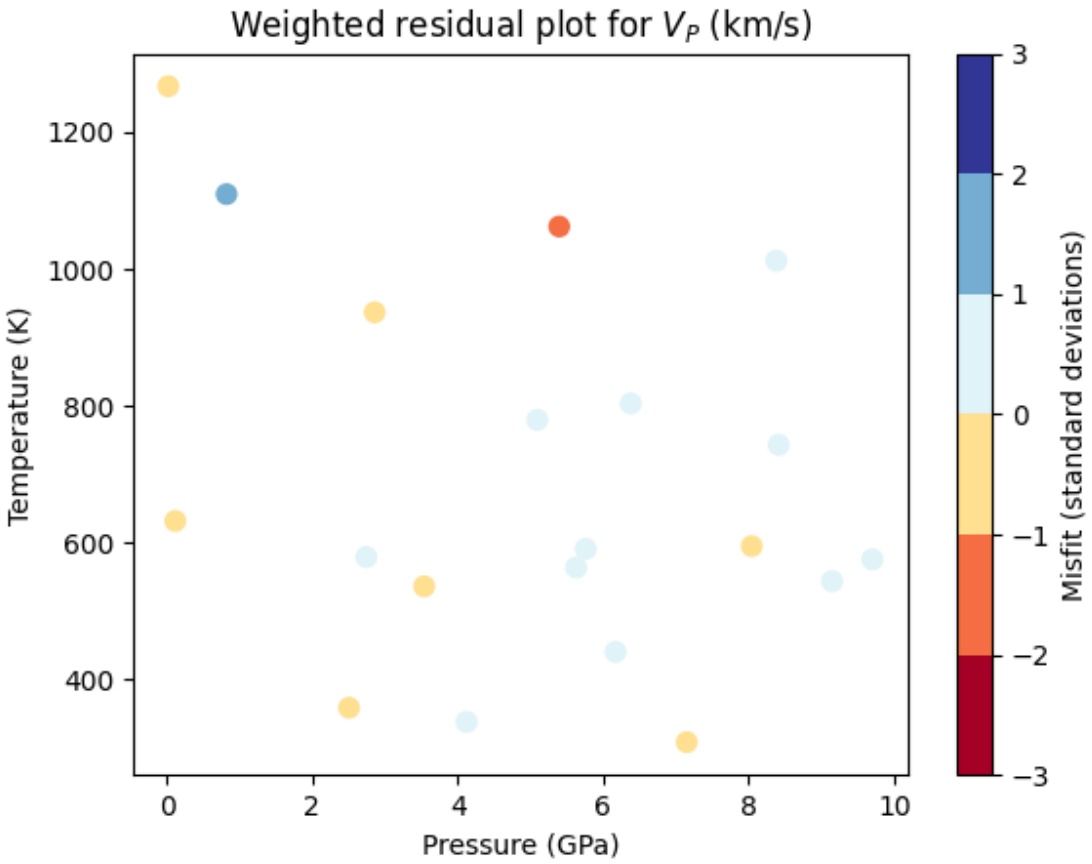


Fig. 5: A P-T plot showing the weighted residuals of each piece of P-wave velocity data.

5.3.11 example_equilibrate

This example demonstrates how BurnMan may be used to calculate the equilibrium state for an assemblage of a fixed bulk composition given two constraints. Each constraint has the form [<constraint type>, <constraint>], where <constraint type> is one of the strings: P, T, S, V, X, PT_ellipse, phase_fraction, or phase_composition. The <constraint> object should either be a float or an array of floats for P, T, S, V (representing the desired pressure, temperature, entropy or volume of the material). If the constraint type is X (a generic constraint on the solution vector) then the constraint c is represented by the following equality: np.dot(c[0], x) - c[1]. If the constraint type is PT_ellipse, the equality is given by norm(([P, T] - c[0])/c[1]) - 1. The constraint_type phase_fraction assumes a tuple of the phase object (which must be one of the phases in the burnman.Composite) and a float or vector corresponding to the phase fractions. Finally, a phase_composition constraint has the format (site_names, n, d, v), where site names dictates the sites involved in the equality constraint. The equality constraint is given by n*x/d*x = v, where x are the site occupancies and n and d are fixed vectors of site coefficients. So, one could for example choose a constraint ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) which would correspond to equal amounts Mg and Fe on the A site.

This script provides a number of examples, which can be turned on and off with a series of boolean variables. In order of complexity:

- run_aluminosilicates: Creates the classic aluminosilicate diagram involving univariate reactions between andalusite, sillimanite and kyanite.
- run_ordering: Calculates the state of order of Jennings and Holland (2015) orthopyroxene in the simple en-fs binary at 1 bar.
- run_gt_solvus: Demonstrates the shape of the pyrope-grossular solvus.
- run_fper_ol: Calculates the equilibrium Mg-Fe partitioning between ferropericlase and olivine.
- run_fixed_ol_composition: Calculates the composition of wadsleyite in equilibrium with olivine of a fixed composition at a fixed pressure.
- run_upper_mantle: Calculates the equilibrium compositions and phase proportions for an ol-opx-gt composite in an NCFMAS bulk composition.
- run_lower_mantle: Calculates temperatures and assemblage properties along an isentrope in the lower mantle. Includes calculations of the post-perovskite-in and bridgmanite-out lines.
- run_olivine_polymorphs: Produces a P-T pseudosection for a fo90 composition.

Uses:

- *Mineral databases*
- `burnman.Composite`
- `burnman.equilibrate()`

Resulting figures:

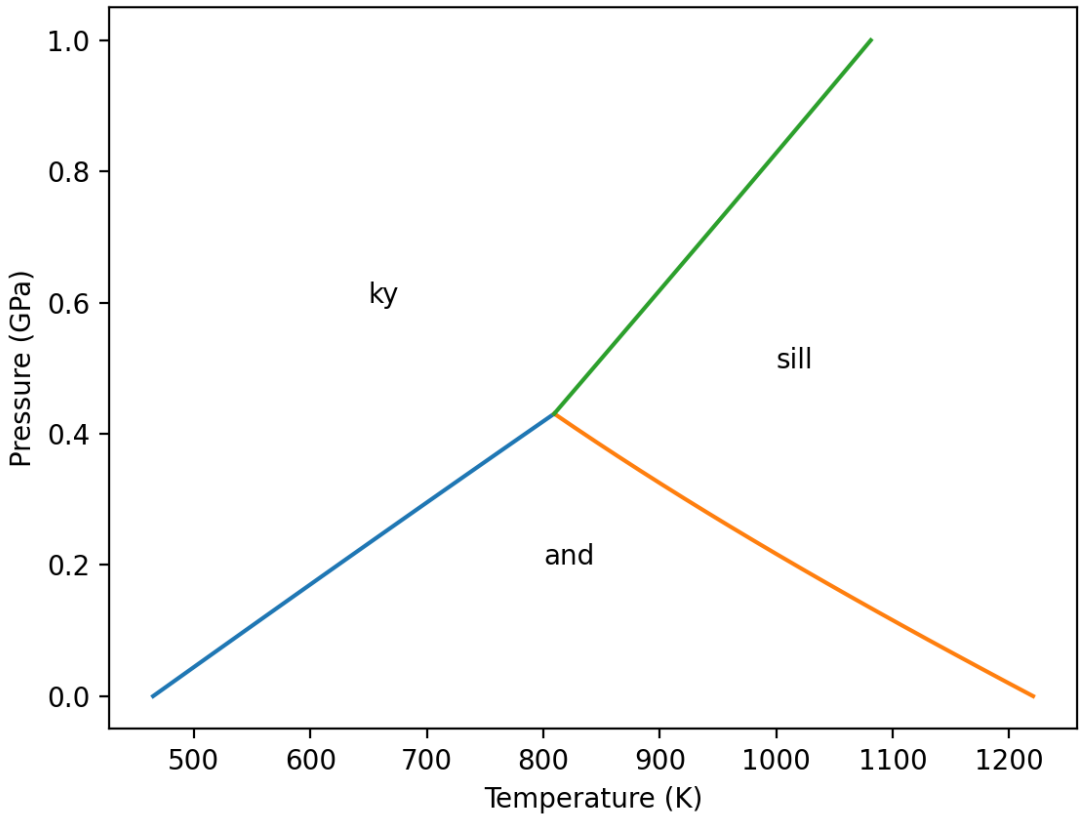


Fig. 6: The classic aluminosilicate diagram.

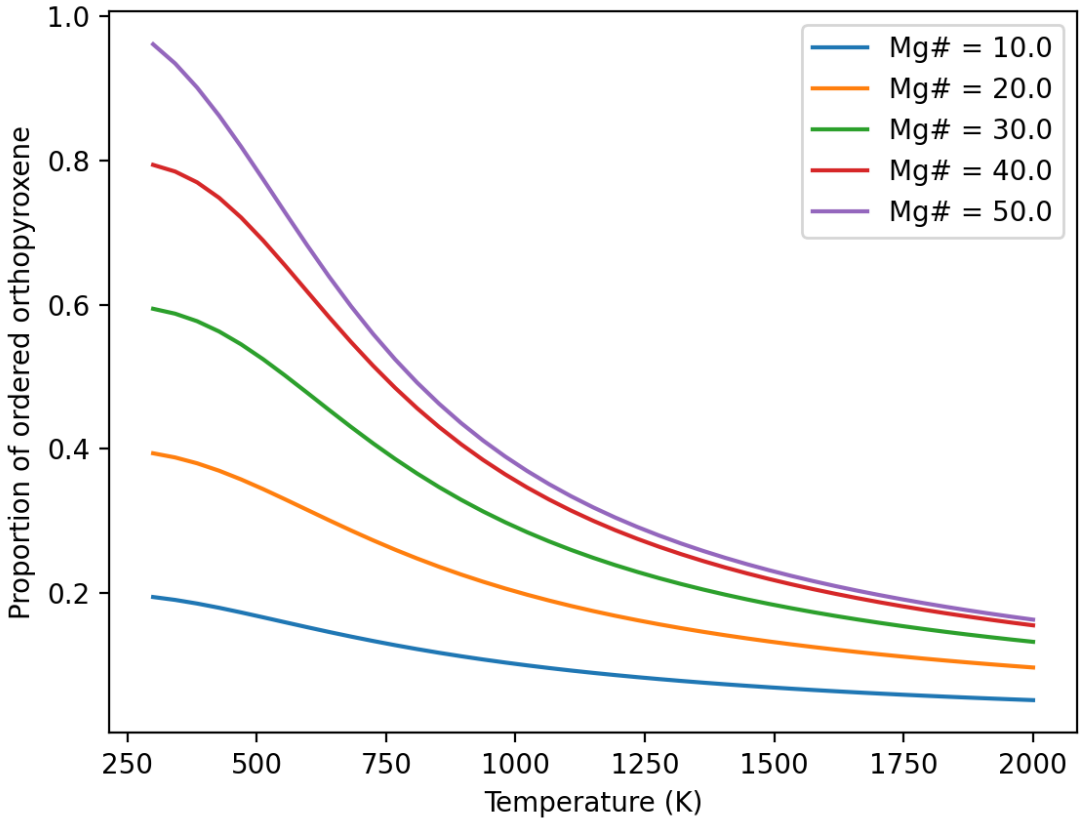


Fig. 7: Ordering in two site orthopyroxene.

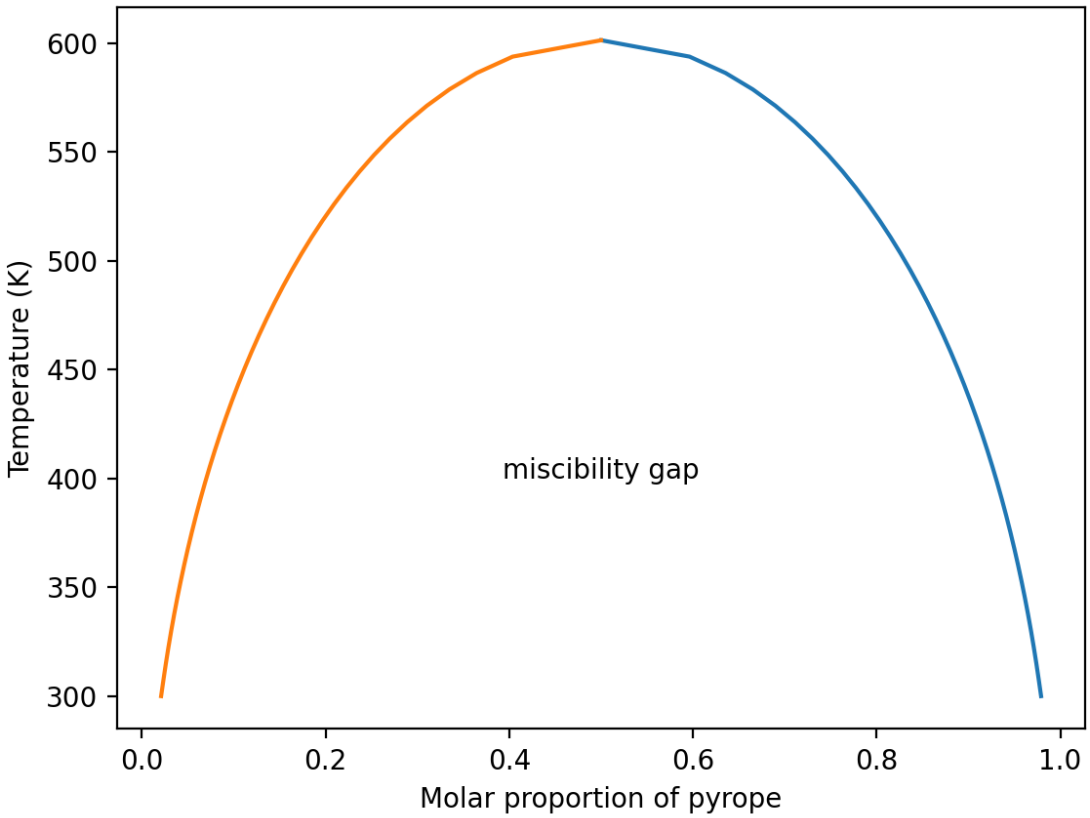


Fig. 8: Miscibility in the pyrope-grossular garnet system

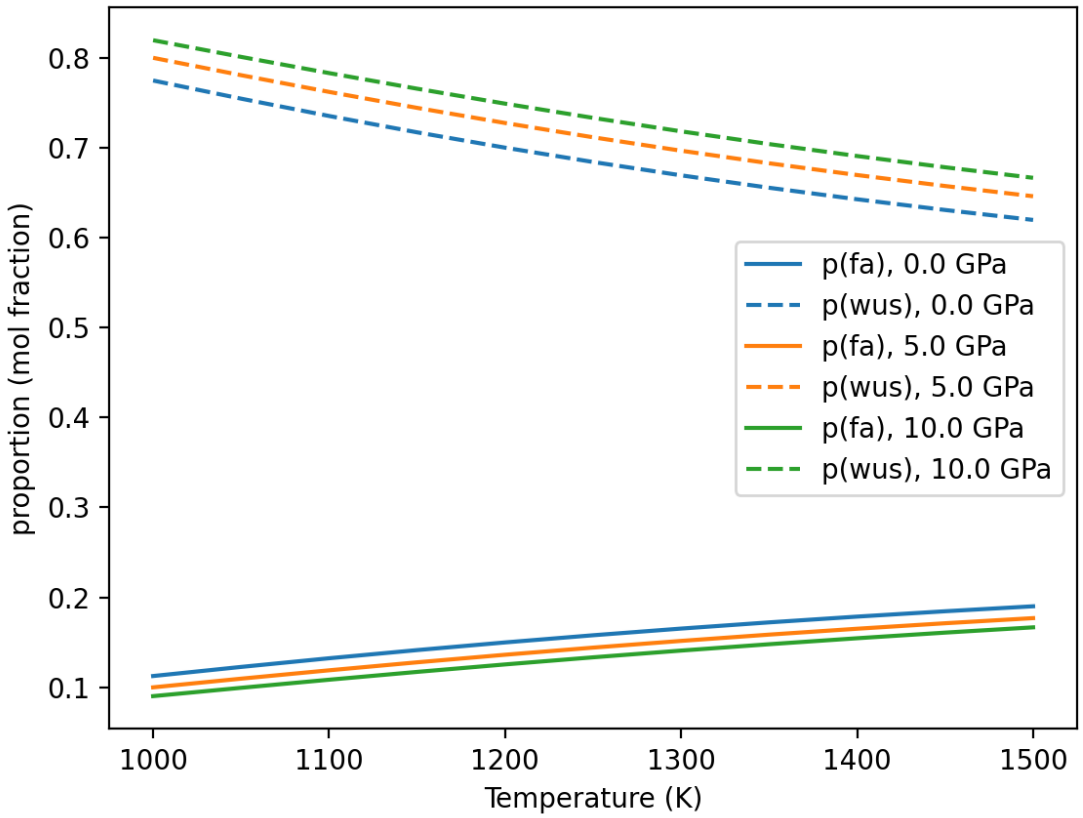


Fig. 9: Mg-Fe partitioning between olivine and ferropericlase.

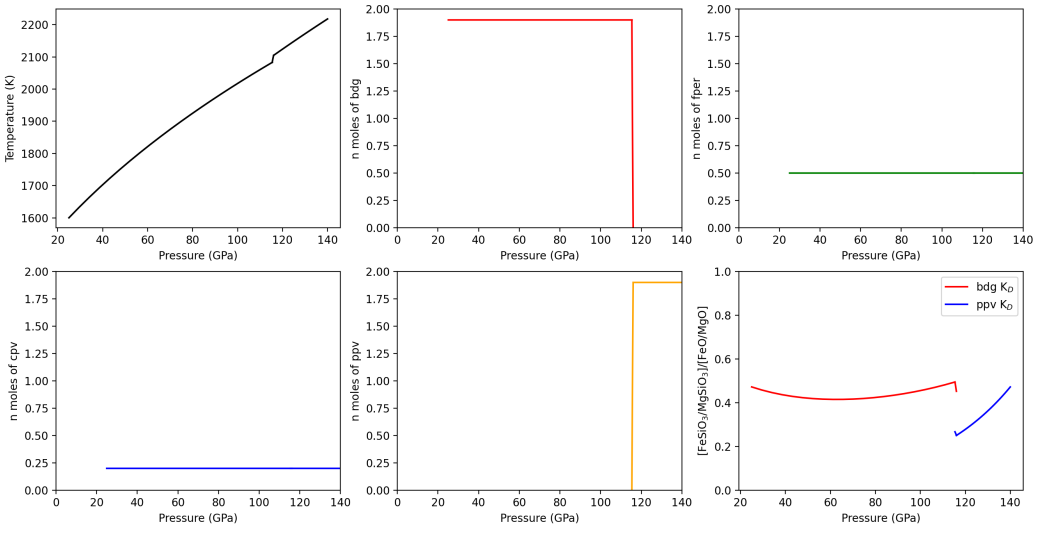


Fig. 10: Phase equilibria in the lower mantle.

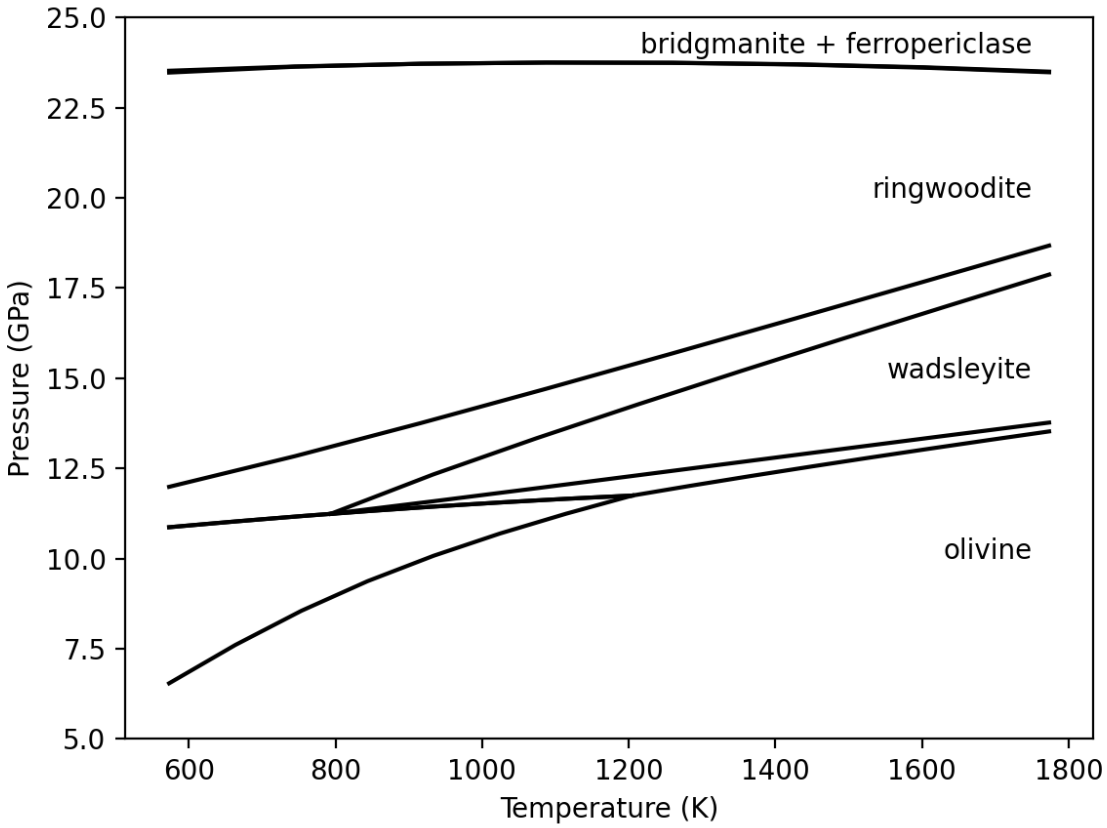


Fig. 11: A P-T pseudosection for a composition of Fe0.2Mg1.8SiO4 (fo90).

5.3.12 example_olivine_binary

This example demonstrates how BurnMan may be used to calculate the equilibrium binary phase diagram for the three olivine polymorphs (olivine, wadsleyite and ringwoodite).

The calculations use the equilibrate function. Unlike the examples in example_equilibrate.py, which are constrained to a fixed bulk composition, the bulk composition is allowed to vary along the vector [n_Mg - n_Fe].

Uses:

- *Mineral databases*
- `burnman.Composite`
- `burnman.equilibrate()`

Resulting figures:

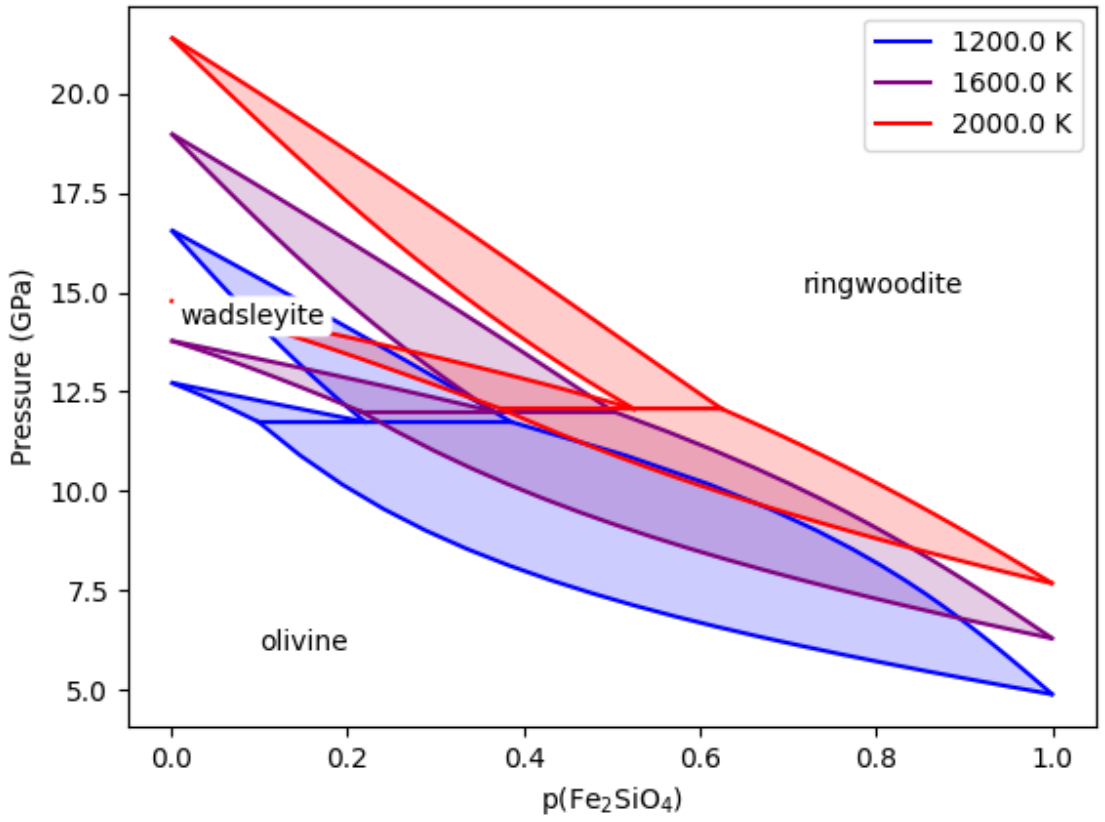


Fig. 12: The olivine polymorph binary phase diagram using the thermodynamic models of Stixrude and Lithgow-Bertelloni (2011).

5.4 Reproducing Cottaar, Heister, Rose and Unterborn (2014)

In this section we include the scripts that were used for all computations and figures in the 2014 BurnMan paper: Cottaar, Heister, Rose & Unterborn (2014) [CHRU14]

5.4.1 paper_averaging

This script reproduces [CHRU14], Figure 2.

This example shows the effect of different averaging schemes. Currently four averaging schemes are available: 1. Voight-Reuss-Hill 2. Voight averaging 3. Reuss averaging 4. Hashin-Shtrikman averaging

See [WDOConnell76] for explanations of each averaging scheme.

requires: - geotherms - compute seismic velocities

teaches: - averaging

5.4.2 paper_benchmark

This script reproduces the benchmark in [CHRU14], Figure 3.

5.4.3 paper_fit_data

This script reproduces [CHRU14] Figure 4.

This example demonstrates BurnMan's functionality to fit thermoelastic data to both 2nd and 3rd orders using the EoS of the user's choice at 300 K. User's must create a file with P, T and V_s. See input_minphys/ for example input files.

requires: - compute seismic velocities

teaches: - averaging

`contrib.CHRU2014.paper_fit_data.`**`calc_shear_velocities`**(*G_0*, *Gprime_0*, *mineral*, *pressures*)

`contrib.CHRU2014.paper_fit_data.`**`error`**(*guess*, *test_mineral*, *pressures*, *obs_vs*)

5.4.4 paper_incorrect_averaging

This script reproduces [CHRU14], Figure 5. Attempt to reproduce Figure 6.12 from [Mur13]

5.4.5 paper_opt_pv

This script reproduces [CHRU14], Figure 6. Vary the amount perovskite vs. ferropericlase and compute the error in the seismic data against PREM.

requires: - creating minerals - compute seismic velocities - geotherms - seismic models - seismic comparison

teaches: - compare errors between models - loops over models

5.4.6 paper_onefit

This script reproduces [CHRU14], Figure 7. It shows an example for a best fit for a pyrolitic model within mineralogical error bars.

5.4.7 paper_uncertain

This script reproduces [CHRU14], Figure 8. It shows the sensitivity of the velocities to various mineralogical parameters.

5.5 Misc or work in progress

5.5.1 example_grid

This example shows how to evaluate seismic quantities on a P, T grid.

5.5.2 example_woutput

This example explains how to perform the basic i/o of BurnMan. A method of calculation is chosen, a composite mineral/material (see example_composition.py for explanation of this process) is created in the class "rock," finally a geotherm is created and seismic velocities calculated.

Post-calculation, the results are written to a simple text file to plot/manipulate at the user's whim.

requires: - creating minerals - compute seismic velocities - geotherms

teaches: - output computed seismic data to file

CHAPTER

SIX

AUTOGENERATED FULL API

6.1 Materials

Burnman operates on materials (type *Material*) most prominently in the form of minerals (*Mineral*) and composites (*Composite*).

6.1.1 Material Base Class

class burnman.**Material**

Bases: object

Base class for all materials. The main functionality is unroll() which returns a list of objects of type *Mineral* and their molar fractions. This class is available as burnman.Material.

The user needs to call set_method() (once in the beginning) and set_state() before querying the material with unroll() or density().

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

set_method(*method*)

Set the averaging method. See *Averaging Schemes* for details.

Note: Needs to be implemented in derived classes.

to_string()

Returns a human-readable name of this material. The default implementation will return the name of the class, which is a reasonable default.

Returns

A human-readable name of the material.

Return type

str

debug_print(*indent=''*)

Print a human-readable representation of this Material.

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

set_state(*pressure*, *temperature*)

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

copy()

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns

A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type
numpy.array, n-dimensional

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_mass

Returns molar mass of the mineral.

Note: Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property molar_volume

Returns molar volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
float

property density

Returns the density of this material.

Note: Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
float

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Note: Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property isothermal_compressibility

Returns isothermal compressibility of the mineral (or inverse isothermal bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property adiabatic_compressibility

Returns adiabatic compressibility of the mineral (or inverse adiabatic bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property shear_modulus

Returns shear modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
float

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameter [unitless].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property energy

Alias for *molar_internal_energy()*

property helmholtz

Alias for *molar_helmholtz()*

property gibbs

Alias for *molar_gibbs()*

property V

Alias for *molar_volume()*

property rho

Alias for *density()*

property S

Alias for *molar_entropy()*

property H

Alias for *molar_enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property beta_T

Alias for *isothermal_compressibility()*

property beta_S

Alias for *adiabatic_compressibility()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property G

Alias for *shear_modulus()*

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property gr

Alias for *grueneisen_parameter()*

property alpha

Alias for *thermal_expansivity()*

property C_v

Alias for *molar_heat_capacity_v()*

property C_p

Alias for *molar_heat_capacity_p()*

6.1.2 Perple_X Class

class burnman.**PerplexMaterial**(*tab_file*, *name='Perple_X material'*)

Bases: *Material*

This is the base class for a PerpleX material. States of the material can only be queried after setting the pressure and temperature using set_state().

Instances of this class are initialised with a 2D PerpleX tab file. This file should be in the standard format (as output by werami), and should have columns with the following names: 'rho,kg/m3', 'alpha,1/K', 'beta,1/bar', 'Ks,bar', 'Gs,bar', 'v0,km/s', 'vp,km/s', 'vs,km/s', 's,J/K/kg', 'h,J/kg', 'cp,J/K/kg', 'V,J/bar/mol'. The order of these names is not important.

Properties of the material are determined by linear interpolation from the PerpleX grid. They are all returned in SI units on a molar basis, even though the PerpleX tab file is not in these units.

This class is available as `burnman.PerplexMaterial`.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

set_state()

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property molar_volume

Returns molar volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
float

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Note: Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

property shear_modulus

Returns shear modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs free energy in [J/mol].

Return type
float

`property molar_mass`

Returns molar mass of the mineral.

Note: Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

`property density`

Returns the density of this material.

Note: Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
float

`property molar_internal_energy`

Returns the molar internal energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
float

`property molar_helmholtz`

Returns the molar Helmholtz free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz free energy in [J/mol].

Return type
float

property isothermal_compressibility

Returns isothermal compressibility of the mineral (or inverse isothermal bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property adiabatic_compressibility

Returns adiabatic compressibility of the mineral (or inverse adiabatic bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
float

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameter [unitless].

Return type
float

property C_p
Alias for *molar_heat_capacity_p()*

property C_v
Alias for *molar_heat_capacity_v()*

property G
Alias for *shear_modulus()*

property H
Alias for *molar_enthalpy()*

property K_S
Alias for *adiabatic_bulk_modulus()*

property K_T
Alias for *isothermal_bulk_modulus()*

property P
Alias for *pressure()*

property S
Alias for *molar_entropy()*

property T
Alias for *temperature()*

property V
Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss
Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss
Alias for *adiabatic_compressibility()*

property alpha
Alias for *thermal_expansivity()*

property beta_S
Alias for *adiabatic_compressibility()*

property beta_T
Alias for *isothermal_compressibility()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns

Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_method(*method*)

Set the averaging method. See *Averaging Schemes* for details.

Note: Needs to be implemented in derived classes.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns a human-readable name of this material. The default implementation will return the name of the class, which is a reasonable default.

Returns
A human-readable name of the material.

Return type
str

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

6.1.3 Minerals

6.1.3.1 Endmembers

class burnman.**Mineral**(*params=None*, *property_modifiers=None*)

Bases: *Material*

This is the base class for all minerals. States of the mineral can only be queried after setting the pressure and temperature using set_state(). The method for computing properties of the material is set using set_method(). This is done during initialisation if the param 'equation_of_state' has been defined. The method can be overridden later by the user.

This class is available as `burnman.Mineral`.

If deriving from this class, set the properties in self.params to the desired values. For more complicated materials you can overwrite set_state(), change the params and then call set_state() from this class.

All the material parameters are expected to be in plain SI units. This means that the elastic moduli should be in Pascals and NOT Gigapascals, and the Debye temperature should be in K not C. Additionally, the reference volume should be in m^3/(mol molecule) and not in unit cell volume and 'n' should be the number of atoms per molecule. Frequently in the literature the reference volume is given in Angstrom^3 per unit cell. To convert this to m^3/(mol of molecule) you should multiply by 10^(-30) * N_a / Z, where N_a is Avogadro's number and Z is the number of formula units per unit cell. You can look up Z in many places, including www.mindat.org

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

set_method(*equation_of_state*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

to_string()

Returns the name of the mineral class

debug_print(*indent=''*)

Print a human-readable representation of this Material.

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns

A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type

tuple

set_state()

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].

- **temperature** (*float*) – The desired temperature in [K].

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_volume

Returns molar volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
float

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Note: Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

property shear_modulus

Returns shear modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
float

property formula

Returns the chemical formula of the Mineral class

property molar_mass

Returns molar mass of the mineral.

Note: Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property density

Returns the density of this material.

Note: Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
float

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property isothermal_compressibility

Returns isothermal compressibility of the mineral (or inverse isothermal bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property adiabatic_compressibility

Returns adiabatic compressibility of the mineral (or inverse adiabatic bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
float

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameter [unitless].

Return type
float

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns

Temperature in [K].

Return type

float

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns

Isochoric heat capacity in [J/K/mol].

Return type
float

6.1.3.2 Solutions

class burnman.**Solution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all solutions. Site occupancies, endmember activities and the constant and pressure and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state().

This class is available as *burnman.Solution*. It uses an instance of burnman.SolutionModel to calculate interaction terms between endmembers.

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and P derivatives in J/K/mol and m^3/mol.

The parameters are relevant to all solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (burnman.SolutionModel) – The SolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters
molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns

A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

burnman.**SolidSolution**

alias of *Solution*

class burnman.**ElasticSolution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all Elastic solutions. Site occupancies, endmember activities and the constant and volume and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state() and set_composition.

This class is available as *burnman.ElasticSolution*. It uses an instance of *burnman.ElasticSolutionModel* to calculate interaction terms between endmembers.

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and V derivatives in J/K/mol and Pa/mol.

The parameters are relevant to all Elastic solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.

- **solution_model** (*burnman.ElasticSolutionModel*) – The ElasticSolution-Model object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property partial_gibbs

Returns endmember partial molar Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial molar volumes [m^3/mol]. Property specific to solutions.

property partial_entropies

Returns endmember partial molar entropies [J/K/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property molar_helmholtz

Returns molar Helmholtz energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_pressure

Returns excess pressure of the solution [Pa]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

`to_string()`

Returns the name of the mineral class

`unroll()`

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

`property v_p`

Alias for *p_wave_velocity()*

`property v_phi`

Alias for *bulk_sound_velocity()*

`property v_s`

Alias for *shear_wave_velocity()*

burnman.**ElasticSolidSolution**

alias of *ElasticSolution*

6.1.3.3 Mineral helpers

class burnman.classes.mineral_helpers.**HelperSpinTransition**(*transition_pressure*, *ls_mat*, *hs_mat*)

Bases: *Composite*

Helper class that makes a mineral that switches between two materials (for low and high spin) based on some transition pressure [Pa]

`debug_print(indent='')`

Print a human-readable representation of this Material.

set_state(*pressure*, *temperature*)

Update the material to the given pressure [Pa] and temperature [K].

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus

Returns adiabatic bulk modulus of the mineral [Pa] Aliased with self.K_S

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility

Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_S

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

property bulk_sound_velocity

Returns bulk sound speed of the composite [m/s] Aliased with self.v_phi

chemical_potential(*components=None*)

Returns the chemical potentials of the currently defined components in the composite. Raises an exception if the assemblage is not equilibrated.

Parameters

components (*list of dictionaries*) – List of formulae of the desired components. If not specified, the method uses the components specified by a previous call to set_components.

Returns

The chemical potentials of the desired components in the equilibrium composite.

Return type

numpy.array of floats

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the composite.

copy()

property density

Compute the density of the composite based on the molar volumes and masses Aliased with self.rho

property dependent_element_indices

The element indices not included in the independent list.

property elements

A list of the elements which could be contained in the composite, returned in the IUPAC element order.

property endmember_formulae

A list of the formulae in the composite.

property endmember_names

A list of the endmember names contained in the composite. Mineral names are returned as given in Mineral.name. Solution endmember names are given in the format *Mineral.name in Solution.name*.

property endmember_partial_gibbs

Returns the partial Gibbs energies for all the endmember minerals in the Composite

property endmembers_per_phase

A list of integers corresponding to the number of endmembers stored within each phase.

property energy

Alias for *molar_internal_energy()*

property equilibrated

Returns True if the reaction affinities are all zero within a given tolerance given by self.equilibrium_tolerance.

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property formula

Returns molar chemical formula of the composite

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the composite [unitless] Aliased with self.gr

property helmholtz

Alias for *molar_helmholtz()*

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts)

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the composite [Pa] Aliased with self.K_T

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility

Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_T

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property molar_enthalpy

Returns enthalpy of the mineral [J] Aliased with self.H

property molar_entropy

Returns enthalpy of the mineral [J] Aliased with self.S

property molar_gibbs

Returns molar Gibbs free energy of the composite [J/mol] Aliased with self.gibbs

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the composite [J/K/mol] Aliased with self.C_p

property molar_heat_capacity_v

Returns molar_heat capacity at constant volume of the composite [J/K/mol] Aliased with self.C_v

property molar_helmholtz

Returns molar Helmholtz free energy of the mineral [J/mol] Aliased with self.helmholtz

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol] Aliased with self.energy

property molar_mass

Returns molar mass of the composite [kg/mol]

property molar_volume

Returns molar volume of the composite [m^3/mol] Aliased with self.V

property n_elements

Returns the total number of distinct elements which might be in the composite.

property n_endmembers

Returns the number of endmembers in the composite.

property n_reactions

The number of reactions in reaction_basis.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property p_wave_velocity

Returns P wave speed of the composite [m/s] Aliased with self.v_p

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

property reaction_affinities

Returns the affinities corresponding to each reaction in reaction_basis

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property reaction_basis_as_strings

Returns a list of string representations of all the reactions in reaction_basis.

property reduced_stoichiometric_array

The stoichiometric array including only the independent elements

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_averaging_scheme(*averaging_scheme*)

Set the averaging scheme for the moduli in the composite. Default is set to VoigtReussHill, when Composite is initialized.

set_components(*components*)

Sets the components and components_array attributes of the composite material. The components attribute is a list of dictionaries containing the chemical formulae of the components. The components_array attribute is a 2D numpy array describing the linear transformation between endmember amounts and component amounts.

The components do not need to be linearly independent, not do they need to form a complete basis set for the composite. However, it must be possible to obtain the composition of each component from a linear sum of the endmember compositions of the composite. For example, if the composite was composed of MgSiO3 and Mg2SiO4, SiO2 would be a valid component, but Si would not. The method raises an exception if any of the chemical potentials are not defined by the assemblage.

Parameters

components (*list of dictionaries*) – List of formulae of the components.

set_fractions(*fractions*, *fraction_type='molar'*)

Change the fractions of the phases of this Composite. Resets cached properties

Parameters

- **fractions** – list or numpy array of floats molar or mass fraction for each phase.
- **fraction_type** – 'molar' or 'mass' specify whether molar or mass fractions are specified.

set_method(*method*)

set the same equation of state method for all the phases in the composite

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral [Pa] Aliased with self.G

property shear_wave_velocity

Returns shear wave speed of the composite [m/s] Aliased with self.v_s

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_matrix

An sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the composite [1/K] Aliased with self.alpha

to_string()

return the name of the composite

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

6.1.3.4 Anisotropic materials

class burnman.**AnisotropicMaterial**(*rho*, *cijs*)

Bases: *Material*

A base class for anisotropic elastic materials. The base class is initialised with a density and a full isentropic stiffness tensor in Voigt notation. It can then be interrogated to find the values of different properties, such as bounds on seismic velocities. There are also several functions which can be called to calculate properties along directions oriented with respect to the isentropic elastic tensor.

See [MHS11] and https://docs.materialsproject.org/methodology/elasticity/ for mathematical descriptions of each function.

property isentropic_stiffness_tensor

property full_isentropic_stiffness_tensor

property isentropic_compliance_tensor

property full_isentropic_compliance_tensor

property density

Returns the density of this material.

Note: Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
float

property isentropic_bulk_modulus_voigt

Returns
The Voigt bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_reuss

Returns
The Reuss bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_shear_modulus_voigt

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_reuss

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the laterial response to loading.

Return type
float

christoffel_tensor(*propagation_direction*)

Returns
The Christoffel tensor from an elastic stiffness tensor and a propagation direction for a seismic wave relative to the stiffness tensor: T_ik = C_ijkl n_j n_l.

Return type
float

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction

relative to the stiffness tensor [1/Pa]. :rtype: float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction

relative to the stiffness tensor [Pa]. :rtype: float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given

shear direction relative to the stiffness tensor [Pa]. :rtype: float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response

directions relative to the stiffness tensor [unitless]. :rtype: float

wave_velocities(*propagation_direction*)

Returns
The compressional wave velocity, and two

shear wave velocities in a given propagation direction [m/s]. :rtype: list, containing the wave speeds and directions

of particle motion relative to the stiffness tensor

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility

Returns adiabatic compressibility of the mineral (or inverse adiabatic bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
float

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type
numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameter [unitless].

Return type
float

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Note: Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility

Returns isothermal compressibility of the mineral (or inverse isothermal bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property molar_enthalpy

Returns molar enthalpy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
float

property molar_mass

Returns molar mass of the mineral.

Note: Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property molar_volume

Returns molar volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
float

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property p_wave_velocity

Returns P wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
float

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_method(*method*)

Set the averaging method. See *Averaging Schemes* for details.

Note: Needs to be implemented in derived classes.

set_state(*pressure*, *temperature*)

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
float

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

to_string()

Returns a human-readable name of this material. The default implementation will return the name of the class, which is a reasonable default.

Returns
A human-readable name of the material.

Return type
str

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns

A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

class burnman.**AnisotropicMineral**(*isotropic_mineral*, *cell_parameters*, *anisotropic_parameters*, *psi_function=None*, *orthotropic=None*)

Bases: *Mineral*, *AnisotropicMaterial*

A class implementing the anisotropic mineral equation of state described in [Myhill22]. This class is derived from both Mineral and AnisotropicMaterial, and inherits most of the methods from these classes.

Instantiation of an AnisotropicMineral takes three required arguments; a reference Mineral (i.e. a standard isotropic mineral which provides volume as a function of pressure and temperature), cell_parameters, which give the lengths of the molar cell vectors and the angles between them (see *cell_parameters_to_vectors()*), and an anisotropic parameters object, which should be either a 4D array of anisotropic parameters or a dictionary of parameters which describe the anisotropic behaviour of the mineral. For a description of the physical meaning of the parameters in the 4D array, please refer to the code or to the original paper.

If the user chooses to define their parameters as a dictionary, they must also provide a function to the psi_function argument that describes how to compute the tensors Psi, dPsidf and dPsidPth (in Voigt form). The function arguments should be f, Pth and params, in that order. The output variables Psi, dPsidf and dPsidth must be returned in that order in a tuple. The user should also explicitly state whether the material is orthotropic or not by supplying a boolean to the orthotropic argument.

States of the mineral can only be queried after setting the pressure and temperature using set_state().

This class is available as `burnman.AnisotropicMineral`.

All the material parameters are expected to be in plain SI units. This means that the elastic moduli should be in Pascals and NOT Gigapascals. Additionally, the cell parameters should be in m/(mol formula unit) and not in unit cell lengths. To convert unit cell lengths given in Angstrom to molar cell parameters you should multiply by 10^(-10) * (N_a / Z)^1/3, where N_a is Avogadro's number and Z is the number of formula units per unit cell. You can look up Z in many places, including www.mindat.org.

Finally, it is assumed that the unit cell of the anisotropic material is aligned in a particular way relative to the coordinate axes (the anisotropic_parameters are defined relative to the coordinate axes). The crystallographic a-axis is assumed to be parallel to the first spatial coordinate axis, and the crystallographic b-axis is assumed to be perpendicular to the third spatial coordinate axis.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

standard_psi_function(*f*, *Pth*, *params*)

set_state()

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property deformation_gradient_tensor

Returns
The deformation gradient tensor describing the deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature).

Return type
numpy.array (2D)

property unrotated_cell_vectors

Returns
The vectors of the cell [m] constructed from one mole of formula units after deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type
numpy.array (2D)

property deformed_coordinate_frame

Returns
The orientations of the three spatial coordinate axes after deformation of the mineral [m]. For orthotropic minerals, this is equal to the identity matrix, as hydrostatic stresses only induce rotations in monoclinic and triclinic crystals.

Return type
numpy.array (2D)

property rotation_matrix

Returns
The matrix required to rotate the properties of the deformed mineral into the deformed coordinate frame. For orthotropic minerals, this is equal to the identity matrix.

Return type
numpy.array (2D)

property cell_vectors

Returns
The vectors of the cell constructed from one mole of formula units [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type
numpy.array (2D)

property cell_parameters

Returns
The molar cell parameters of the mineral, given in standard form: [a, b, c, α, β, γ], where the first three floats are the lengths of the vectors in [m] defining the cell constructed from one mole of formula units. The last three floats are angles between vectors (given in radians). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type
numpy.array (1D)

property shear_modulus

Anisotropic minerals do not (in general) have a single shear modulus. This function returns a NotImplementedError. Users should instead consider directly querying the elements in the isothermal_stiffness_tensor or isentropic_stiffness_tensor.

property isothermal_bulk_modulus

Anisotropic minerals do not have a single isothermal bulk modulus. This function returns a NotImplementedError. Users should instead consider either using isothermal_bulk_modulus_reuss, isothermal_bulk_modulus_voigt, or directly querying the elements in the isothermal_stiffness_tensor.

property isentropic_bulk_modulus

Anisotropic minerals do not have a single isentropic bulk modulus. This function returns a NotImplementedError. Users should instead consider either using isentropic_bulk_modulus_reuss, isentropic_bulk_modulus_voigt (both derived from AnisotropicMineral), or directly querying the elements in the isentropic_stiffness_tensor.

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the material.

Note: Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property isothermal_compressibility

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider either using isothermal_compressibility_reuss, isothermal_compressibility_voigt (both derived from AnisotropicMineral), or directly querying the elements in the isothermal_compliance_tensor.

property isentropic_compressibility

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider either using isentropic_compressibility_reuss, isentropic_compressibility_voigt (both derived from AnisotropicMineral), or directly querying the elements in the isentropic_compliance_tensor.

property isothermal_bulk_modulus_voigt

Returns
The Voigt bound on the isothermal bulk modulus in [Pa].

Return type
float

property isothermal_compressibility_reuss

Returns
The Reuss bound on the isothermal compressibility in [1/Pa].

Return type
float

property beta_T

Returns
The Reuss bound on the isothermal compressibility in [1/Pa].

Return type
float

property isothermal_compressibility_voigt

Returns
The Voigt bound on the isothermal compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_reuss

Returns
The Reuss bound on the isentropic compressibility in [1/Pa].

Return type
float

property beta_S

Returns
The Reuss bound on the isentropic compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_voigt

Returns
The Voigt bound on the isentropic compressibility in [1/Pa].

Return type
float

property isothermal_compliance_tensor

Returns
The isothermal compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

property thermal_expansivity_tensor

Returns
The tensor of thermal expansivities [1/K].

Return type
numpy.array (2D)

property isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

property full_isothermal_compliance_tensor

Returns
The isothermal compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property full_isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property full_isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property full_isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property grueneisen_tensor

Returns
The grueneisen tensor [unitless]. This is defined by [BM67] as $\mathbb{C}_{\mathrm{N}ijkl}\alpha_{kl}V/C_P$.

Return type
numpy.array (2D)

property grueneisen_parameter

Returns
The scalar grueneisen parameter [unitless].

Return type
float

property isothermal_compressibility_tensor

Returns
The isothermal compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property isentropic_compressibility_tensor

Returns
The isentropic compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property thermal_stress_tensor

Returns
The change in stress with temperature at constant strain [Pa/K].

Return type
numpy.array (2D)

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility

Returns adiabatic compressibility of the mineral (or inverse adiabatic bulk modulus).

Note: Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
float

christoffel_tensor(*propagation_direction*)

Returns
The Christoffel tensor from an elastic stiffness tensor and a propagation direction for a seismic wave relative to the stiffness tensor: T_ik = C_ijkl n_j n_l.

Return type
float

copy()

debug_print(*indent=''*)
Print a human-readable representation of this Material.

property density
Returns the density of this material.

Note: Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
float

property energy
Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)
Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].

- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type
numpy.array, n-dimensional

property formula

Returns the chemical formula of the Mineral class

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isentropic_bulk_modulus_reuss

Returns
The Reuss bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_voigt

Returns
The Voigt bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the laterial response to loading.

Return type
float

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction

relative to the stiffness tensor [1/Pa]. :rtype: float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response

directions relative to the stiffness tensor [unitless]. :rtype: float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given

shear direction relative to the stiffness tensor [Pa]. :rtype: float

property isentropic_shear_modulus_reuss

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_voigt

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction

relative to the stiffness tensor [Pa]. :rtype: float

property molar_enthalpy

Returns molar enthalpy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
float

property molar_isometric_heat_capacity

Returns
The molar heat capacity at constant strain [J/K/mol].

Return type
float

property molar_mass

Returns molar mass of the mineral.

Note: Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property molar_volume

Returns molar volume of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
float

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_method(*equation_of_state*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property shear_wave_velocity

Returns shear wave speed of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns

Shear wave speed in [m/s].

Return type

float

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Note: Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

wave_velocities(*propagation_direction*)

Returns
The compressional wave velocity, and two

shear wave velocities in a given propagation direction [m/s]. :rtype: list, containing the wave speeds and directions

of particle motion relative to the stiffness tensor

check_standard_parameters(*anisotropic_parameters*)

burnman.**cell_parameters_to_vectors**(*cell_parameters*)

Converts cell parameters to unit cell vectors.

Parameters
cell_parameters (*numpy.array (1D)*) – An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Returns
The three vectors defining the parallelopiped cell [m]. This function assumes that the first cell vector is colinear with the x-axis, and the second is perpendicular to the z-axis, and the third is defined in a right-handed sense.

Return type
numpy.array (2D)

burnman.**cell_vectors_to_parameters**(*M*)

Converts unit cell vectors to cell parameters.

Parameters
M (*numpy.array (2D)*) – The three vectors defining the parallelopiped cell [m]. This function assumes that the first cell vector is colinear with the x-axis, the second is perpendicular to the z-axis, and the third is defined in a right-handed sense.

Returns
An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Return type
numpy.array (1D)

6.1.4 Composites

class burnman.**Composite**(*phases*, *fractions=None*, *fraction_type='molar'*, *name='Unnamed composite'*)

Bases: *Material*

Base class for a composite material. The static phases can be minerals or materials, meaning composite can be nested arbitrarily.

The fractions of the phases can be input as either 'molar' or 'mass' during instantiation, and modified (or initialised) after this point by using set_fractions.

This class is available as `burnman.Composite`.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

set_fractions(*fractions*, *fraction_type='molar'*)

Change the fractions of the phases of this Composite. Resets cached properties

Parameters

- **fractions** – list or numpy array of floats molar or mass fraction for each phase.
- **fraction_type** – 'molar' or 'mass' specify whether molar or mass fractions are specified.

set_method(*method*)

set the same equation of state method for all the phases in the composite

set_averaging_scheme(*averaging_scheme*)

Set the averaging scheme for the moduli in the composite. Default is set to VoigtReussHill, when Composite is initialized.

set_state(*pressure*, *temperature*)

Update the material to the given pressure [Pa] and temperature [K].

debug_print(*indent=''*)

Print a human-readable representation of this Material.

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

`to_string()`
return the name of the composite

`property formula`
Returns molar chemical formula of the composite

`property molar_internal_energy`
Returns molar internal energy of the mineral [J/mol] Aliased with self.energy

`property molar_gibbs`
Returns molar Gibbs free energy of the composite [J/mol] Aliased with self.gibbs

`property molar_helmholtz`
Returns molar Helmholtz free energy of the mineral [J/mol] Aliased with self.helmholtz

`property molar_volume`
Returns molar volume of the composite [m^3/mol] Aliased with self.V

`property molar_mass`
Returns molar mass of the composite [kg/mol]

`property density`
Compute the density of the composite based on the molar volumes and masses Aliased with self.rho

`property molar_entropy`
Returns enthalpy of the mineral [J] Aliased with self.S

`property molar_enthalpy`
Returns enthalpy of the mineral [J] Aliased with self.H

`property isothermal_bulk_modulus`
Returns isothermal bulk modulus of the composite [Pa] Aliased with self.K_T

`property adiabatic_bulk_modulus`
Returns adiabatic bulk modulus of the mineral [Pa] Aliased with self.K_S

`property isothermal_compressibility`
Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_T

`property adiabatic_compressibility`
Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_S

property shear_modulus

Returns shear modulus of the mineral [Pa] Aliased with self.G

property p_wave_velocity

Returns P wave speed of the composite [m/s] Aliased with self.v_p

property bulk_sound_velocity

Returns bulk sound speed of the composite [m/s] Aliased with self.v_phi

property shear_wave_velocity

Returns shear wave speed of the composite [m/s] Aliased with self.v_s

property grueneisen_parameter

Returns grueneisen parameter of the composite [unitless] Aliased with self.gr

property thermal_expansivity

Returns thermal expansion coefficient of the composite [1/K] Aliased with self.alpha

property molar_heat_capacity_v

Returns molar_heat capacity at constant volume of the composite [J/K/mol] Aliased with self.C_v

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the composite [J/K/mol] Aliased with self.C_p

property endmember_partial_gibbs

Returns the partial Gibbs energies for all the endmember minerals in the Composite

property reaction_affinities

Returns the affinities corresponding to each reaction in reaction_basis

property equilibrated

Returns True if the reaction affinities are all zero within a given tolerance given by self.equilibrium_tolerance.

set_components(*components*)

Sets the components and components_array attributes of the composite material. The components attribute is a list of dictionaries containing the chemical formulae of the components. The components_array attribute is a 2D numpy array describing the linear transformation between endmember amounts and component amounts.

The components do not need to be linearly independent, not do they need to form a complete basis set for the composite. However, it must be possible to obtain the composition of each component from a linear sum of the endmember compositions of the composite. For example, if the composite was composed of MgSiO3 and Mg2SiO4, SiO2 would be a valid component, but Si would not. The method raises an exception if any of the chemical potentials are not defined by the assemblage.

Parameters

components (*list of dictionaries*) – List of formulae of the components.

chemical_potential(*components=None*)

Returns the chemical potentials of the currently defined components in the composite. Raises an exception if the assemblage is not equilibrated.

Parameters
components (*list of dictionaries*) – List of formulae of the desired components. If not specified, the method uses the components specified by a previous call to set_components.

Returns
The chemical potentials of the desired components in the equilibrium composite.

Return type
numpy.array of floats

property stoichiometric_matrix

An sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property reaction_basis_as_strings

Returns a list of string representations of all the reactions in reaction_basis.

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts)

property dependent_element_indices

The element indices not included in the independent list.

property reduced_stoichiometric_array

The stoichiometric array including only the independent elements

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the composite.

property endmember_formulae

A list of the formulae in the composite.

property endmember_names

A list of the endmember names contained in the composite. Mineral names are returned as given in Mineral.name. Solution endmember names are given in the format *Mineral.name in Solution.name*.

property endmembers_per_phase

A list of integers corresponding to the number of endmembers stored within each phase.

property elements

A list of the elements which could be contained in the composite, returned in the IUPAC element order.

property n_endmembers

Returns the number of endmembers in the composite.

property n_elements

Returns the total number of distinct elements which might be in the composite.

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

6.1.5 Calibrants

class burnman.**Calibrant**(*calibrant_function*, *calibrant_function_return_type*, *params*)
Bases: object

The base class for a pressure calibrant material.

Parameters

- **calibrant_function** (*function*) – A function that takes either pressure, temperature and a params object as arguments, returning the volume, or takes volume, temperature and a params object, returning the pressure.
- **calibrant_function_return_type** (*str*) – The return type of the calibrant function. Valid values are 'pressure' or 'volume'.
- **params** (*dictionary*) – A dictionary containing the parameters required by the calibrant function.

pressure(*volume*, *temperature*, *VT_covariance=None*)
Returns the pressure of the calibrant as a function of volume, temperature and (optionally) a volume-temperature variance-covariance matrix.

Parameters

- **volume** (*float*) – The volume of the calibrant [m^3/mol].
- **temperature** (*float*) – The temperature of the calibrant [K].
- **VT_covariance** (*2x2 numpy.array*) – The volume-temperature variance-covariance matrix [optional].

Returns
The pressure of the calibrant [Pa] and the pressure-volume-temperature variance-covariance matrix if PT_covariance is provided.

Return type
float, tuple of a float and a numpy.array (3x3)

volume(*pressure*, *temperature*, *PT_covariance=None*)

Returns the volume of the calibrant as a function of pressure, temperature and (optionally) a pressure-temperature variance-covariance matrix.

Parameters

- **pressure** (*float*) – The pressure of the calibrant [Pa].
- **temperature** (*float*) – The temperature of the calibrant [K].
- **PT_covariance** (*2x2 numpy.array*) – The pressure-temperature variance-covariance matrix [optional].

Returns

The volume of the calibrant [m^3/mol] and the volume-pressure-temperature variance-covariance matrix if VT_covariance is provided.

Return type

float, tuple of a float and a numpy.array (3x3)

6.2 Equations of state

6.2.1 Base class

class burnman.eos.**EquationOfState**

Bases: object

This class defines the interface for an equation of state that a mineral uses to determine its properties at a given P, T. In order define a new equation of state, you should define these functions.

All functions should accept and return values in SI units.

In general these functions are functions of pressure, temperature, and volume, as well as a "params" object, which is a Python dictionary that stores the material parameters of the mineral, such as reference volume, Debye temperature, reference moduli, etc.

The functions for volume and density are just functions of temperature, pressure, and "params"; after all, it does not make sense for them to be functions of volume or density.

volume(*pressure*, *temperature*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Molar volume of the mineral $[m^3]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Isothermal bulk modulus of the mineral. $[Pa]$

Return type
float

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Shear modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant pressure of the mineral. $[J/K/mol]$

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Thermal expansivity of the mineral. $[1/K]$

Return type
float

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Gibbs energy of the mineral $[J/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Entropy of the mineral $[J/K/mol]$.

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

validate_parameters(*params*)

The params object is just a dictionary associating mineral physics parameters for the equation of state. Different equation of states can have different parameters, and the parameters may have ranges of validity. The intent of this function is twofold. First, it can check for the existence of the parameters that the equation of state needs, and second, it can check whether the parameters have reasonable values. Unreasonable values will frequently be due to unit issues (e.g., supplying bulk moduli in GPa instead of Pa). In the base class this function does nothing, and an equation of state is not required to implement it. This function will not return anything, though it may raise warnings or errors.

Parameters
params (*dict*) – Dictionary containing material parameters required by the equation of state.

6.2.2 Murnaghan

class burnman.eos.**Murnaghan**

Bases: *EquationOfState*

Base class for the isothermal Murnaghan equation of state, as described in [Mur44].

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$ Currently not included in the Murnghan EOS, so omitted.

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.3 Birch-Murnaghan

6.2.3.1 Base class

class burnman.eos.birch_murnaghan.**BirchMurnaghanBase**

Bases: *EquationOfState*

Base class for the isothermal Birch Murnaghan equation of state. This is third order in strain, and has no temperature dependence. However, the shear modulus is sometimes fit to a second order function, so if this is the case, you should use that. For more see `burnman.birch_murnaghan.BM2` and `burnman.birch_murnaghan.BM3`.

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.3.2 BM2

class burnman.eos.**BM2**

Bases: *BirchMurnaghanBase*

Third order Birch Murnaghan isothermal equation of state. This uses the second order expansion for shear modulus.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type

float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

6.2.3.3 BM3

class burnman.eos.**BM3**

Bases: *BirchMurnaghanBase*

Third order Birch Murnaghan isothermal equation of state. This uses the third order expansion for shear modulus.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)
Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)
Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)
Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)
Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

6.2.3.4 BM4

class burnman.eos.**BM4**

Bases: *EquationOfState*

Base class for the isothermal Birch Murnaghan equation of state. This is fourth order in strain, and has no temperature dependence.

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.4 Vinet

class burnman.eos.**Vinet**

Bases: *EquationOfState*

Base class for the isothermal Vinet equation of state. References for this equation of state are [VFSR86] and [VSFR87]. This equation of state actually predates Vinet by 55 years [Rydberg32], and was investigated further by [StaceyBrennanIrvine81].

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$ Currently not included in the Vinet EOS, so omitted.

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.5 Morse Potential

class burnman.eos.**Morse**

Bases: *EquationOfState*

Class for the isothermal Morse Potential equation of state detailed in [StaceyBrennanIrvine81]. This equation of state has no temperature dependence.

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* [m^3].
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral [J/mol].

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the equation of state [K].
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* [m^3].
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral [J/mol].

Return type
float

6.2.6 Reciprocal K-prime

class burnman.eos.**RKprime**

Bases: *EquationOfState*

Class for the isothermal reciprocal K-prime equation of state detailed in [SD04]. This equation of state is a development of work by [Kea54] and [SD00], making use of the fact that K' typically varies smoothly as a function of P/K, and is thermodynamically required to exceed 5/3 at infinite pressure.

It is worth noting that this equation of state rapidly becomes unstable at negative pressures, so should not be trusted to provide a good *HT-LP* equation of state using a thermal pressure formulation. The negative root of dP/dK can be found at $K/P = K'_\infty - K'_0$, which corresponds to a bulk modulus of $K = K_0(1 - K'_\infty/K'_0)^{K'_0/K'_\infty}$ and a volume of $V = V_0(K'_0/(K'_0 - K'_\infty))^{K'_0/K'^2_\infty} \exp(-1/K'_\infty)$.

This equation of state has no temperature dependence.

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa].

pressure(*temperature*, *volume*, *params*)

Returns pressure $[Pa]$ as a function of volume $[m^3]$.

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus K_s of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters. The value for K'_∞ is thermodynamically bounded between 5/3 and K'_0 [SD04].

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.7 Stixrude and Lithgow-Bertelloni Formulation

6.2.7.1 Base class

class burnman.eos.slb.**SLBBase**

Bases: *EquationOfState*

Base class for the finite strain-Mie-Grueneiesen-Debye equation of state detailed in [SLB05]. For the most part the equations are all third order in strain, but see further the `burnman.slb.SLB2` and `burnman.slb.SLB3` classes.

volume_dependent_q(*x*, *params*)

Finite strain approximation for q, the isotropic volume strain derivative of the grueneisen parameter.

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter $[unitless]$

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

6.2.7.2 SLB2

class burnman.eos.**SLB2**

Bases: *SLBBase*

SLB equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter $[unitless]$

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

volume_dependent_q(*x*, *params*)

Finite strain approximation for q, the isotropic volume strain derivative of the grueneisen parameter.

6.2.7.3 SLB3

class burnman.eos.**SLB3**

Bases: *SLBBase*

SLB equation of state with third order finite strain expansion for the shear modulus (this should be preferred, as it is more thermodynamically consistent.)

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter $[unitless]$

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

volume_dependent_q(*x*, *params*)

Finite strain approximation for q, the isotropic volume strain derivative of the grueneisen parameter.

6.2.8 Mie-Grüneisen-Debye

6.2.8.1 Base class

class burnman.eos.mie_grueneisen_debye.**MGDBase**

Bases: *EquationOfState*

Base class for a generic finite-strain Mie-Grueneisen-Debye equation of state. References for this can be found in many places, such as Shim, Duffy and Kenichi (2002) and Jackson and Rigden (1996). Here we mostly follow the appendices of Matas et al (2007). Of particular note is the thermal correction to the shear modulus, which was developed by Hama and Suito (1998).

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume (EQ B6)

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] EQ B7

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B8

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B11

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol]

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume [J/K/mol]

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ D6

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3] EQ B7

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

6.2.8.2 MGD2

class burnman.eos.**MGD2**

Bases: *MGDBase*

MGD equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ D6

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume (EQ B6)

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B8

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume [J/K/mol]

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3] EQ B7

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B11

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] EQ B7

6.2.8.3 MGD3

class burnman.eos.**MGD3**

Bases: *MGDBase*

MGD equation of state with third order finite strain expansion for the shear modulus (this should be preferred, as it is more thermodynamically consistent.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ D6

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume (EQ B6)

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B8

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume [J/K/mol]

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3] EQ B7

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ B11

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] EQ B7

6.2.9 Modified Tait

class burnman.eos.**MT**

Bases: *EquationOfState*

Base class for the generic modified Tait equation of state. References for this can be found in [Huang-Chow74] and [HollandPowell11] (followed here).

An instance "m" of a Mineral can be assigned this equation of state with the command m.set_method('mt') (or by initialising the class with the param equation_of_state = 'mt').

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3]

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T of the mineral. $[Pa]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Not implemented in the Modified Tait EoS. $[Pa]$ Returns 0. Could potentially apply a fixed Poissons ratio as a rough estimate.

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy $\mathcal{S}$ of the mineral. $[J/K/mol]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy $\mathcal{E}$ of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return a very large number. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[1/K]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Since this equation of state does not contain temperature effects, simply return zero. $[unitless]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

6.2.10 Holland and Powell Formulations

6.2.10.1 HP_TMT (2011 solid formulation)

class burnman.eos.**HP_TMT**

Bases: *EquationOfState*

Base class for the thermal equation of state based on the generic modified Tait equation of state (class MT), as described in [HollandPowell11].

An instance "m" of a Mineral can be assigned this equation of state with the command m.set_method('hp_tmt') (or by initialising the class with the param equation_of_state = 'hp_tmt'

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] EQ 12

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3] EQ B7

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume.

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ 13+2

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Not implemented. Returns 0. Could potentially apply a fixed Poissons ratio as a rough estimate.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol].

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]. This function replaces -Pth in EQ 13+1 with P-Pth for non-ambient temperature

molar_heat_capacity_p0(*temperature*, *params*)

Returns heat capacity at ambient pressure as a function of temperature [J/K/mol]. Cp = a + bT + cT^-2 + dT^-0.5 in [HollandPowell11].

molar_heat_capacity_p_einstein(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume, using the C_v and Einstein model [J/K/mol] WARNING: Only for comparison with internally self-consistent C_p

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy [J/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

6.2.10.2 HP_TMTL (2011 liquid formulation)

class burnman.eos.**HP_TMTL**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98], but with the Modified Tait as the static part, as described in [HollandPowell11].

An instance "m" of a Mineral can be assigned this equation of state with the command m.set_method('hp_tmtL') (or by initialising the class with the param equation_of_state = 'hp_tmtL'

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume.

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Not implemented. Returns 0. Could potentially apply a fixed Poissons ratio as a rough estimate.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol].

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

molar_heat_capacity_p0(*temperature*, *params*)

Returns heat capacity at ambient pressure as a function of temperature [J/K/mol] Cp = a + bT + cT^-2 + dT^-0.5 in [HollandPowell98].

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy [J/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

6.2.10.3 HP98 (1998 formulation)

class burnman.eos.**HP98**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98].

An instance "m" of a Mineral can be assigned this equation of state with the command m.set_method('hp98') (or by initialising the class with the param equation_of_state = 'hp98'

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume.

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Not implemented. Returns 0. Could potentially apply a fixed Poissons ratio as a rough estimate.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol].

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

molar_heat_capacity_p0(*temperature*, *params*)

Returns heat capacity at ambient pressure as a function of temperature [J/K/mol] Cp = a + bT + cT^-2 + dT^-0.5 in [HollandPowell98].

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy [J/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Internal energy of the mineral $[J/mol]$.

Return type

float

6.2.11 De Koker Solid and Liquid Formulations

6.2.11.1 DKS_S (Solid formulation)

class burnman.eos.**DKS_S**

Bases: *EquationOfState*

Base class for the finite strain solid equation of state detailed in [deKokerKarkiStixrude13] (supplementary materials).

volume_dependent_q(*x*, *params*)

Finite strain approximation for q, the isotropic volume strain derivative of the grueneisen parameter.

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter $[unitless]$

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

6.2.11.2 DKS_L (Liquid formulation)

class burnman.eos.**DKS_L**

Bases: *EquationOfState*

Base class for the finite strain liquid equation of state detailed in [deKokerKarkiStixrude13] (supplementary materials).

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

volume(*pressure*, *temperature*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Molar volume of the mineral $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter. $[unitless]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$ Zero for fluids

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol]

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol]

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Internal energy of the mineral $[J/mol]$.

Return type

float

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

6.2.12 Anderson and Ahrens (1994)

class burnman.eos.**AA**

Bases: *EquationOfState*

Class for the :math`E-V-S` liquid metal EOS detailed in [AndersonAhrens94]. Internal energy (E) is first calculated along a reference isentrope using a fourth order BM EoS (V_0, KS, KS', KS''), which gives volume as a function of pressure, coupled with the thermodynamic identity:

$-\partial E/\partial V|_S = P.$

The temperature along the isentrope is calculated via

$\partial(\ln T)/\partial(\ln \rho)|_S = \gamma$

which gives:

$T_S/T_0 = \exp(\int(\gamma/\rho)d\rho)$

The thermal effect on internal energy is calculated at constant volume using expressions for the kinetic, electronic and potential contributions to the volumetric heat capacity, which can then be integrated with respect to temperature:

$\partial E/\partial T|_V = C_V$

$\partial E/\partial S|_V = T$

We note that [AndersonAhrens94] also include a detailed description of the Gruneisen parameter as a function of volume and energy (Equation 15), and use this to determine the temperature along the principal isentrope (Equations B1-B10) and the thermal pressure away from that isentrope (Equation 23). However, this expression is inconsistent with the equation of state away from the principal isentrope. Here we choose to calculate the thermal pressure and Grueneisen parameter thus:

1) As energy and entropy are defined by the equation of state at any temperature and volume, pressure can be found by via the expression:

$\partial E/\partial V|_S = P$

2) The Grueneisen parameter can now be determined as $\gamma = V\partial P/\partial E|_V$

To reiterate: away from the reference isentrope, the Grueneisen parameter calculated using these expressions is *not* equal to the (thermodynamically inconsistent) analytical expression given by [AndersonAhrens94].

A final note: the expression for Λ (Equation 17). does not reproduce Figure 5. We assume here that the figure matches the model actually used by [AndersonAhrens94], which has the form: $F(-325.23 + 302.07(\rho/\rho_0) + 30.45(\rho/\rho_0)^{0.4})$.

volume_dependent_q(*x*, *params*)

Finite strain approximation for q, the isotropic volume strain derivative of the grueneisen parameter.

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter $[unitless]$

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus. $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$ Zero for a liquid

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$ Currently found by numerical differentiation (1/V * dV/dT)

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol] E + PV

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy at the pressure and temperature of the mineral [J/mol]

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Returns the enthalpy at the pressure and temperature of the mineral [J/mol] E + PV

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Helmholtz free energy at the pressure and temperature of the mineral [J/mol] E - TS

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

6.2.13 CoRK

class burnman.eos.**CORK**

Bases: *EquationOfState*

Class for the CoRK equation of state detailed in [HP91]. The CoRK EoS is a simple virial-type extension to the modified Redlich-Kwong (MRK) equation of state. It was designed to compensate for the tendency of the MRK equation of state to overestimate volumes at high pressures and accommodate the volume behaviour of coexisting gas and liquid phases along the saturation curve.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns grueneisen parameter [unitless] as a function of pressure, temperature, and volume.

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] Eq. 7 in Holland and Powell, 1991

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3]. EQ 13+2

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Not implemented. Returns 0. Could potentially apply a fixed Poissons ratio as a rough estimate.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant volume at the pressure, temperature, and volume [J/K/mol].

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K] Replace -Pth in EQ 13+1 with P-Pth for non-ambient temperature

molar_heat_capacity_p0(*temperature*, *params*)

Returns heat capacity at ambient pressure as a function of temperature [J/K/mol] Cp = a + bT + cT^-2 + dT^-0.5 in Holland and Powell, 2011

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume [J/K/mol]

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns adiabatic bulk modulus [Pa] as a function of pressure [Pa], temperature [K], and volume [m^3].

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Entropy of the mineral $[J/K/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

6.2.14 Brosh et al. (2007)

class burnman.eos.**BroshCalphad**

Bases: *EquationOfState*

Class for the high pressure CALPHAD equation of state by [BMS07].

volume(*pressure*, *temperature*, *params*)
Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns the isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

adiabatic_bulk_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns the adiabatic bulk modulus of the mineral. $[Pa]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns the shear modulus G of the mineral. $[Pa]$

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the internal energy of the mineral. $[J/mol]$

gibbs_free_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy of the mineral. $[J/mol]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy of the mineral. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the molar isobaric heat capacity $[J/K/mol]$. For now, this is calculated by numerical differentiation.

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns the volumetric thermal expansivity $[1/K]$. For now, this is calculated by numerical differentiation.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Returns the grueneisen parameter. This is a dependent thermodynamic variable in this equation of state.

calculate_transformed_parameters(*params*)

This function calculates the "c" parameters of the [BMS07] equation of state.

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

6.3 Solution models

Solution objects in BurnMan are instances of one of two classes: type *Solution* (alias *SolidSolution*) and type *ElasticSolution* (alias *ElasticSolidSolution*). The *Solution* class implements commonly used models (in petrology). Excess properties are defined relative to the endmember properties at fixed pressure and temperature. The formulations are defined with interaction parameters such as excess energies, volumes and entropies.

The *ElasticSolution* class instead defines excess properties are relative to the endmember properties at fixed volume and temperature. Such models have their roots in atom-scale considerations; mixing of two instances of the same lattice type requires deformation (local lattice distortions), that can be considered to induce a local chemical stress. Therefore, volume may be a more useful independent variable than pressure. For more details, see [Myhill18].

The *Solution* and *ElasticSolution* classes both accept several methods which define the properties of the solution.

6.3.1 Base classes

class burnman.**Solution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all solutions. Site occupancies, endmember activities and the constant and pressure and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state().

This class is available as *burnman.Solution*. It uses an instance of burnman.SolutionModel to calculate interaction terms between endmembers.

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and P derivatives in J/K/mol and m^3/mol.

The parameters are relevant to all solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (burnman.SolutionModel) – The SolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

class burnman.**ElasticSolution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all Elastic solutions. Site occupancies, endmember activities and the constant and volume and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state() and set_composition.

This class is available as *burnman.ElasticSolution*. It uses an instance of *burnman.ElasticSolutionModel* to calculate interaction terms between endmembers.

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and V derivatives in J/K/mol and Pa/mol.

The parameters are relevant to all Elastic solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.ElasticSolutionModel*) – The ElasticSolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property partial_gibbs

Returns endmember partial molar Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial molar volumes [m^3/mol]. Property specific to solutions.

property partial_entropies

Returns endmember partial molar entropies [J/K/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property molar_helmholtz

Returns molar Helmholtz energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_pressure

Returns excess pressure of the solution [Pa]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *molar_heat_capacity_p()*

property C_v

Alias for *molar_heat_capacity_v()*

property G

Alias for *shear_modulus()*

property H

Alias for *molar_enthalpy()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property K_T

Alias for *isothermal_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *molar_entropy()*

property T

Alias for *temperature()*

property V

Alias for *molar_volume()*

property adiabatic_bulk_modulus_reuss

Alias for *adiabatic_bulk_modulus()*

property adiabatic_compressibility_reuss

Alias for *adiabatic_compressibility()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *adiabatic_compressibility()*

property beta_T

Alias for *isothermal_compressibility()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property energy

Alias for *molar_internal_energy()*

evaluate(*vars_list*, *pressures*, *temperatures*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

Array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i].

Return type

numpy.array, n-dimensional

property gibbs

Alias for *molar_gibbs()*

property gr

Alias for *grueneisen_parameter()*

property helmholtz

Alias for *molar_helmholtz()*

property isothermal_bulk_modulus_reuss

Alias for *isothermal_bulk_modulus()*

property isothermal_compressibility_reuss

Alias for *isothermal_compressibility()*

property pressure

Returns current pressure that was set with *set_state()*.

Note: Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [5.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

Note: Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

Note: Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

class burnman.**ElasticSolutionModel**

Bases: object

This is the base class for an Elastic solution model, intended for use in defining solutions and performing thermodynamic calculations on them. All minerals of type *burnman.Solution* use a solution model for defining how the endmembers in the solution interact.

A user wanting a new solution model should define the functions included in the base class. All of the functions in the base class return zero, so if the user-defined solution model does not implement them, they essentially have no effect, and the Helmholtz energy and pressure of a solution will be equal to the weighted arithmetic averages of the different endmember values.

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess Helmholtz energy.

Return type

float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess pressure of the solution.

Return type

float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of the solution.

Return type

float

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

6.3.2 Mechanical solution

class burnman.classes.solutionmodel.**MechanicalSolution**(*endmembers*)

Bases: SolutionModel

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of mixing; the total gibbs free energy of the solution is equal to the dot product of the molar gibbs free energies and molar fractions of the constituent materials.

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess volume of the solution.

Return type

float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess entropy of the solution.

Return type

float

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess enthalpy of the solution.

Return type
float

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].

- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess()
Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()
Returns the excess V/K_T of the solution model at its current state

alphaV_excess()
Returns the excess alpha*V of the solution model at its current state

class burnman.classes.elasticsolutionmodel.**ElasticMechanicalSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of mixing; the total Helmholtz energy of the solution is equal to the dot product of the molar Helmholtz energies and molar fractions of the constituent materials.

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)
Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)
Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

6.3.3 Ideal solution

class burnman.classes.solutionmodel.**IdealSolution**(*endmembers*)

Bases: SolutionModel

A class representing an ideal solution model. Calculates the excess gibbs free energy and entropy due to configurational entropy. Excess internal energy and volume are equal to zero.

The multiplicity of each type of site in the structure is allowed to change linearly as a function of endmember proportions. This class is therefore equivalent to the entropic part of a Temkin-type model [Tem45].

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess()

Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()

Returns the excess V/K_T of the solution model at its current state

alphaV_excess()

Returns the excess alpha*V of the solution model at its current state

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticIdealSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

A class representing an ideal solution model. Calculates the excess Helmholtz energy and entropy due to configurational entropy. Excess internal energy and volume are equal to zero.

The multiplicity of each type of site in the structure is allowed to change linearly as a function of endmember proportions. This class is therefore equivalent to the entropic part of a Temkin-type model [Tem45].

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess pressure of each endmember.

Return type

numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess enthalpy of the solution.

Return type

float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

6.3.4 Asymmetric regular solution

class burnman.classes.solutionmodel.**AsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *IdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Gibbs energy is given by the expression:

$$\mathcal{G}_{\text{excess}} = \alpha^T p(\phi^T W \phi)$$

α is a vector of van Laar parameters governing asymmetry in the excess properties.

$$\phi_i = \frac{\alpha_i p_i}{\sum_{k=1}^{n} \alpha_k p_k}, W_{ij} = \frac{2w_{ij}}{\alpha_i + \alpha_j} \text{for i<j}$$

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial Gibbs free energy of each endmember.

Return type

numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial entropy of each endmember.

Return type

numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess()

Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()

Returns the excess V/K_T of the solution model at its current state

alphaV_excess()

Returns the excess alpha*V of the solution model at its current state

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticAsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*

Bases: *ElasticIdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Helmholtz energy is given by the expression:

$$\mathcal{F}_{\text{excess}} = \alpha^T p(\phi^T W \phi)$$

α is a vector of van Laar parameters governing asymmetry in the excess properties.

$$\phi_i = \frac{\alpha_i p_i}{\sum_{k=1}^{n} \alpha_k p_k}, W_{ij} = \frac{2w_{ij}}{\alpha_i + \alpha_j} \text{for i<j}$$

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

6.3.5 Symmetric regular solution

class burnman.classes.solutionmodel.**SymmetricRegularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *AsymmetricRegularSolution*

Solution model implementing the symmetric regular solution model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class.

Cp_excess()

Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()

Returns the excess V/K_T of the solution model at its current state

activities(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

alphaV_excess()

Returns the excess alpha*V of the solution model at its current state

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess enthalpy of the solution.

Return type

float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].

- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each

endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

class burnman.classes.elasticsolutionmodel.**ElasticSymmetricRegularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*)

Bases: *ElasticAsymmetricRegularSolution*

Solution model implementing the symmetric regular solution model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class.

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

6.3.6 Subregular solution

class burnman.classes.solutionmodel.**SubregularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *volume_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *IdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess nonconfigurational Gibbs energy is given by the expression:

$$\mathcal{G}_{\text{excess}} = \sum_i \sum_{j>i} (p_i p_j^2 W_{ij} + p_j p_i^2 W_{ji} + \sum_{k>j>i} p_i p_j p_k W_{ijk})$$

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **volume_interaction** (*list of list of lists*) – The binary endmember interaction volumes. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **volume_ternary_terms** (*list of lists*) – The ternary interaction volumes. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess()

Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()

Returns the excess V/K_T of the solution model at its current state

alphaV_excess()

Returns the excess alpha*V of the solution model at its current state

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess enthalpy of the solution.

Return type

float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess entropy of the solution.

Return type
float

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticSubregularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *pressure_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *ElasticIdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess conconfigurational Helmholtz energy is given by the expression:

$$\mathcal{F}_{\text{excess}} = \sum_i \sum_{j>i} (p_i p_j^2 W_{ij} + p_j p_i^2 W_{ji} + \sum_{k>j>i} p_i p_j p_k W_{ijk})$$

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **pressure_interaction** (*list of list of lists*) – The binary endmember interaction pressures. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **pressure_ternary_terms** (*list of lists*) – The ternary interaction pressures. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]

- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

6.3.7 Function solution

class burnman.classes.solutionmodel.**FunctionSolution**(*endmembers*, *excess_gibbs_function*)

Bases: *IdealSolution*

Solution model implementing a generalized solution model. The extensive excess nonconfigurational Gibbs energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Gibbs energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **excess_gibbs_function** (*function*) – The nonconfigurational Gibbs energy function with arguments pressure, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Gibbs energy must also double.

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

excess_partial_gibbs_free_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess()
Returns the excess heat capacity of the solution model at its current state

VoverKT_excess()
Returns the excess V/K_T of the solution model at its current state

alphaV_excess()
Returns the excess alpha*V of the solution model at its current state

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)
Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)
Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_free_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticFunctionSolution**(*endmembers*, *excess_helmholtz_function*)

Bases: *ElasticIdealSolution*

Solution model implementing a generalized elastic solution model. The extensive excess nonconfigurational Helmholtz energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Helmholtz energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **excess_helmholtz_function** (*function*) – The nonconfigurational Helmholtz energy function with arguments volume, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Helmholtz energy must also double.

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_enthalpy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

6.4 Solution tools

burnman.tools.solution.**transform_solution_to_new_basis**(*solution*, *new_basis*, *n_mbrs=None*, *solution_name=None*, *endmember_names=None*, *molar_fractions=None*)

Transforms a solution model from one endmember basis to another. Returns a new Solution object.

Parameters

- **solution** (*burnman.Solution* object) – The original solution object.
- **new_basis** (*2D numpy array*) – The new endmember basis, given as amounts of the old endmembers.
- **n_mbrs** (*float, optional*) – The number of endmembers in the new solution (defaults to the length of new_basis).

- **solution_name** (*str, optional*) – A name corresponding to the new solution.
- **endmember_names** (*list of str, optional*) – A list corresponding to the names of the new endmembers.
- **molar_fractions** (*numpy.array, optional*) – Fractions of the new endmembers in the new solution.

Returns
The transformed solution.

Return type
burnman.Solution object

6.5 Compositions

6.5.1 Base class

class burnman.**Composition**(*composition_dictionary*, *unit_type='mass'*, *normalize=False*)

Bases: object

Class for a composition object, which can be used to store, modify and renormalize compositions, and also convert between mass, molar and atomic amounts. Weight is provided as an alias for mass, as we assume that only Earthlings will use this software.

This class is available as burnman.Composition.

renormalize(*unit_type*, *normalization_component*, *normalization_amount*)

Change the total amount of material in the composition to satisfy a given normalization condition (mass, weight, molar, or atomic)

Parameters

- **unit_type** (*str*) – 'mass', 'weight', 'molar' or 'atomic' Unit type with which to normalize the composition
- **normalization_component** (*str*) – Component/element on which to renormalize. String must either be one of the components/elements already in the composition, or have the value 'total'.
- **normalization_amount** (*float*) – Amount of component in the renormalised composition.

add_components(*composition_dictionary*, *unit_type*)

Add (or remove) components from the composition. The components are added to the current state of the (mass, weight or molar) composition; if the composition has been renormalised, then this should be taken into account.

Parameters

- **composition_dictionary** (*dictionary*) – Components to add, and their amounts.

- **unit_type** (*str*) – 'mass', 'weight' or 'molar'. Unit type of the components to be added.

change_component_set(*new_component_list*)

Change the set of basis components without changing the bulk composition.

Will raise an exception if the new component set is invalid for the given composition.

Parameters
new_component_list (*list of strings*) – New set of basis components.

property weight_composition

An alias for mass composition [kg].

property molar_composition

Returns the molar composition as a counter [moles]

property atomic_composition

Returns the atomic composition as a counter [moles]

composition(*unit_type*)

Helper function to return the composition in the desired type.

Parameters
unit_type (*str*) – One of 'mass', 'weight', 'molar' and 'atomic'.

Returns
Mass (weight), molar or atomic composition.

Return type
OrderedCounter

print(*unit_type*, *significant_figures=1*, *normalization_component='total'*, *normalization_amount=None*)

Pretty-print function for the composition This does not renormalize the Composition object itself, only the printed values.

Parameters

- **unit_type** (*str*) – 'mass', 'weight', 'molar' or 'atomic' Unit type in which to print the composition.
- **significant_figures** (*int*) – Number of significant figures for each amount.
- **normalization_component** (*str*) – Component/element on which to renormalize. String must either be one of the components/elements already in composite, or have the value 'total'. (default = 'total')
- **normalization_amount** (*float*) – Amount of component in the renormalised composition. If not explicitly set, no renormalization will be applied.

6.5.2 Utility functions

burnman.classes.composition.**file_to_composition_list**(*fname*, *unit_type*, *normalize*)

Takes an input file with a specific format and returns a list of compositions (and associated comments) contained in that file.

Parameters

- **fname** (*str*) – Path to ascii file containing composition data. Lines beginning with a hash are not read. The first read-line of the datafile contains a list of tab or space-separated components (e.g. FeO or SiO2), followed by the word Comment. Following lines are lists of floats with the amounts of each component. After the component amounts, the user can write anything they like in the Comment section.
- **unit_type** (*str*) – 'mass', 'weight' or 'molar' Specify whether the compositions in the file are given as mass (weight) or molar amounts.
- **normalize** – If False, absolute numbers of moles/grams of component are stored, otherwise the component amounts of returned compositions will sum to one (until Composition.renormalize() is used).

:type normalize : bool

6.5.3 Fitting functions

burnman.optimize.composition_fitting.**fit_composition_to_solution**(*solution*, *fitted_variables*, *variable_values*, *variable_covariances*, *variable_conversions=None*, *normalize=True*)

Takes a Solution object and a set of variable names and associates values and covariances and finds the molar fractions of the solution which provide the best fit (in a least-squares sense) to the variable values.

The fitting applies appropriate non-negativity constraints (i.e. no species can have a negative occupancy on a site).

Parameters

- **solution** (*burnman.Solution*) – The solution to use in the fitting procedure.
- **fitted_variables** (*list of str*) – A list of the variables used to find the best-fit molar fractions of the solution. These should either be elements such as "Fe", site_species such as "Fef_B" which would correspond to a species labelled Fef on the second site, or user-defined variables which are arithmetic sums of elements and/or site_species defined in "variable_conversions".
- **variable_values** (*numpy.array*) – Numerical values of the fitted variables. These should be given as amounts; they do not need to be normalized.

- **variable_covariances** (*2D numpy.array*) – Covariance matrix of the variables.
- **variable_conversions** (*dict of dict, or None*) – A dictionary converting any user-defined variables into an arithmetic sum of element and site-species amounts. For example, {'Mg_equal': {'Mg_A': 1., 'Mg_B': -1.}}, coupled with Mg_equal = 0 would impose a constraint that the amount of Mg would be equal on the first and second site in the solution.
- **normalize** (*bool*) – If True, normalizes the optimized molar fractions to sum to unity.

Returns
Optimized molar fractions, corresponding covariance matrix and the weighted residual.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

burnman.optimize.composition_fitting.**fit_phase_proportions_to_bulk_composition**(*phase_compositions*, *bulk_composition*)

Performs weighted constrained least squares on a set of phase compositions to find the amount of those phases that best-fits a given bulk composition.

The fitting applies appropriate non-negativity constraints (i.e. no phase can have a negative abundance in the bulk).

Parameters

- **phase_compositions** (*2D numpy.array*) – The composition of each phase. Can be in weight or mole amounts.
- **bulk_composition** (*numpy.array*) – The bulk composition of the composite. Must be in the same units as the phase compositions.

Returns
Optimized molar fractions, corresponding covariance matrix and the weighted residual.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

6.6 Polytopes

Often in mineral physics, solutions are subject to a set of linear constraints. For example, the set of valid site-occupancies in solution models are constrained by positivity and fixed sum constraints (the amount of each chemical species must be greater than or equal to zero, the sites in the structure are present in fixed ratios). Similarly, the phase amounts in a composite must be more than or equal to zero, and the compositions of each phase must sum to the bulk composition of the composite.

Geometrically, linear equality and inequality constraints can be visualised as a polytope (an n-dimensional) polyhedron. There are several situations where it is convenient to be able to interrogate such objects to

understand the space of validity. In BurnMan, we make use of the module pycddlib to create polytope objects. We also provide a number of tools for common chemically-relevant operations.

6.6.1 Base class

class burnman.**MaterialPolytope**(*equalities*, *inequalities*, *number_type='fraction'*, *return_fractions=False*, *independent_endmember_occupancies=None*)

Bases: object

A class that can be instantiated to create pycddlib polytope objects. These objects can be interrogated to provide the vertices satisfying the input constraints.

This class is available as burnman.polytope.MaterialPolytope.

set_return_type(*return_fractions=False*)

Sets the return_type for the polytope object. Also deletes the cached endmember_occupancies property.

Parameters

return_fractions (*bool*) – Choose whether the generated polytope object should return fractions or floats.

property raw_vertices

Returns a list of the vertices of the polytope without any postprocessing. See also endmember_occupancies.

property limits

Return the limits of the polytope (the set of bounding inequalities).

property n_endmembers

Return the number of endmembers (the number of vertices of the polytope).

property endmember_occupancies

Return the endmember occupancies (a processed list of all of the vertex locations).

property independent_endmember_occupancies

Return an independent set of endmember occupancies (a linearly-independent set of vertex locations)

property endmembers_as_independent_endmember_amounts

Return a list of all the endmembers as a linear sum of the independent endmembers.

property independent_endmember_polytope

Returns the polytope expressed in terms of proportions of the independent endmembers. The polytope involves the first n-1 independent endmembers. The last endmember proportion makes the sum equal to one.

property independent_endmember_limits

Gets the limits of the polytope as a function of the independent endmembers.

subpolytope_from_independent_endmember_limits(*limits*)

Returns a smaller polytope by applying additional limits to the amounts of the independent endmembers.

subpolytope_from_site_occupancy_limits(*limits*)

Returns a smaller polytope by applying additional limits to the individual site occupancies.

grid(*points_per_edge=2*, *unique_sorted=True*, *grid_type='independent endmember proportions'*, *limits=None*)

Create a grid of points which span the polytope.

Parameters

- **points_per_edge** (*int*) – Number of points per edge of the polytope.
- **unique_sorted** (*bool*) – The gridding is done by splitting the polytope into a set of simplices. This means that points will be duplicated along vertices, faces etc. If unique_sorted is True, this function will sort and make the points unique. This is an expensive operation for large polytopes, and may not always be necessary.
- **grid_type** (*str*) – Whether to grid the polytope in terms of independent endmember proportions or site occupancies. Choices are 'independent endmember proportions' or 'site occupancies'
- **limits** (*numpy.array (2D)*) – Additional inequalities restricting the gridded area of the polytope.

Returns
A list of points gridding the polytope.

Return type
numpy.array (2D)

6.6.2 Polytope tools

burnman.tools.polytope.**solution_polytope_from_charge_balance**(*charges*, *charge_total*, *return_fractions=False*)

Creates a polytope object from a list of the charges for each species on each site and the total charge for all site-species.

Parameters

- **charges** (*2D list of floats*) – 2D list containing the total charge for species j on site i, including the site multiplicity. So, for example, a solution with the site formula [Mg,Fe]3[Mg,Al,Si]2Si3O12 would have the following list: [[6., 6.], [4., 6., 8.]].
- **charge_total** (*float*) – The total charge for all site-species per formula unit. The example given above would have charge_total = 12.

- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**solution_polytope_from_endmember_occupancies**(*endmember_occupancies*, *return_fractions=False*)

Creates a polytope object from a list of independent endmember occupancies.

Parameters

- **endmember_occupancies** (*2D numpy array*) – 2D list containing the site-species occupancies j for endmember i. So, for example, a solution with independent endmembers [Mg]3[Al]2Si3O12, [Mg]3[Mg0.5Si0.5]2Si3O12, [Fe]3[Al]2Si3O12 might have the following array: [[1., 0., 1., 0., 0.], [1., 0., 0., 0.5, 0.5], [0., 1., 1., 0., 0.]], where the order of site-species is Mg_A, Fe_A, Al_B, Mg_B, Si_B.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**composite_polytope_at_constrained_composition**(*composite*, *composition*, *return_fractions=False*)

Creates a polytope object from a Composite object and a composition. This polytope describes the complete set of valid composite endmember amounts that satisfy the compositional constraints.

Parameters

- **composite** (*burnman.Composite* object) – A composite containing one or more Solution and Mineral objects.
- **composition** (*dict*) – A dictionary containing the amounts of each element.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**simplify_composite_with_composition**(*composite*, *composition*)

Takes a composite and a composition, and returns the simplest composite object that spans the solution space at the given composition.

For example, if the composition is given as {'Mg': 2., 'Si': 1.5, 'O': 5.}, and the composite is given as a mix of Mg,Fe olivine and pyroxene solutions, this function will return a composite that only contains the Mg-bearing endmembers.

Parameters

- **composite** (*burnman.Composite* object) – The initial Composite object.
- **composition** (*dict*) – A dictionary containing the amounts of each element.

Returns
The simplified Composite object

Return type
burnman.Composite object

6.7 Averaging Schemes

Given a set of mineral physics parameters and an equation of state we can calculate the density, bulk, and shear modulus for a given phase. However, as soon as we have a composite material (e.g., a rock), the determination of elastic properties become more complicated. The bulk and shear modulus of a rock are dependent on the specific geometry of the grains in the rock, so there is no general formula for its averaged elastic properties. Instead, we must choose from a number of averaging schemes if we want a single value, or use bounding methods to get a range of possible values. The module `burnman.averaging_schemes` provides a number of different average and bounding schemes for determining a composite rock's physical parameters.

6.7.1 Base class

class burnman.averaging_schemes.**AveragingScheme**

Bases: `object`

Base class defining an interface for determining average elastic properties of a rock. Given a list of volume fractions for the different mineral phases in a rock, as well as their bulk and shear moduli, an averaging will give back a single scalar values for the averages. New averaging schemes should define the functions average_bulk_moduli and average_shear_moduli, as specified here.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli K for a composite. This defines the interface for this method, and is not implemented in the base class.

Parameters
volumes (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.

:param bulk_moduli
[List of bulk moduli of each phase in the composite] $[Pa]$.

:param shear_moduli
[List of shear moduli of each phase in the composite] $[Pa]$.

Returns
The average bulk modulus K. $[Pa]$

Return type
float

`average_shear_moduli`(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli G for a composite. This defines the interface for this method, and is not implemented in the base class.

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli of each phase in the composite $[Pa]$.

Returns
The average shear modulus G. $[Pa]$

Return type
float

`average_density`(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

6.7.2 Voigt bound

class burnman.averaging_schemes.**Voigt**

Bases: *AveragingScheme*

Class for computing the Voigt (iso-strain) bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite K with the Voigt (iso-strain) bound, given by:

$$K_V = \Sigma_i V_i K_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns

The Voigt average bulk modulus K_R $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the Voigt (iso-strain) bound, given by:

$$G_V = \Sigma_i V_i G_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns

The Voigt average shear modulus G_V $[Pa]$.

Return type

float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns

Thermal expansivity of the composite α. $[1/K]$

Return type

float

6.7.3 Reuss bound

class burnman.averaging_schemes.**Reuss**

Bases: *AveragingScheme*

Class for computing the Reuss (iso-stress) bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the Reuss (iso-stress) bound, given by:

$$K_R = \left(\Sigma_i \frac{V_i}{K_i} \right)^{-1}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns

The Reuss average bulk modulus K_R $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the Reuss (iso-stress) bound, given by:

$$G_R = \left(\Sigma_i \frac{V_i}{G_i} \right)^{-1}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns

The Reuss average shear modulus G_R $[Pa]$.

Return type

float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns

Density ρ $[kg/m^3]$.

Return type

float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

6.7.4 Voigt-Reuss-Hill average

class burnman.averaging_schemes.**VoigtReussHill**

Bases: *AveragingScheme*

Class for computing the Voigt-Reuss-Hill average for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the Voigt-Reuss-Hill average, given by:

$$K_{VRH} = \frac{K_V + K_R}{2}$$

This is simply a shorthand for an arithmetic average of the bounds given by `burnman.averaging_schemes.voigt` and `burnman.averaging_schemes.reuss`.

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns

The Voigt-Reuss-Hill average bulk modulus K_{VRH} $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli G of a composite with the Voigt-Reuss-Hill average, given by:

$$G_{VRH} = \frac{G_V + G_R}{2}$$

This is simply a shorthand for an arithmetic average of the bounds given by `burnman.averaging_schemes.voigt` and `burnman.averaging_schemes.reuss`.

$$G_V = \Sigma_i V_i G_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns

The Voigt-Reuss-Hill average shear modulus G_{VRH} $[Pa]$.

Return type

float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).

- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

6.7.5 Hashin-Shtrikman upper bound

class burnman.averaging_schemes.**HashinShtrikmanUpper**

Bases: *AveragingScheme*

Class for computing the upper Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the upper Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns
The upper Hashin-Shtrikman average bulk modulus K. $[Pa]$

Return type
float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the upper Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$

- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns
The upper Hashin-Shtrikman average shear modulus G. $[Pa]$

Return type
float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

6.7.6 Hashin-Shtrikman lower bound

class burnman.averaging_schemes.**HashinShtrikmanLower**

Bases: *AveragingScheme*

Class for computing the lower Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements Formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the lower Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$

- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns
The lower Hashin-Shtrikman average bulk modulus K. $[Pa]$

Return type
float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the lower Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns
The lower Hashin-Shtrikman average shear modulus G. $[Pa]$

Return type
float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

6.7.7 Hashin-Shtrikman arithmetic average

class burnman.averaging_schemes.**HashinShtrikmanAverage**

Bases: *AveragingScheme*

Class for computing arithmetic mean of the Hashin-Shtrikman bounds on elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the arithmetic mean of the upper and lower Hashin-Shtrikman bounds.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. Not used in this average. $[Pa]$

Returns

The arithmetic mean of the Hashin-Shtrikman bounds on bulk modulus K $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the arithmetic mean of the upper and lower Hashin-Shtrikman bounds.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. [m^3].
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. Not used in this average. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The arithmetic mean of the Hashin-Shtrikman bounds on shear modulus G $[Pa]$.

Return type

float

average_density(*volumes*, *densities*)

Average the densities of a composite, given a list of volume fractions and densitites. This is

implemented in the base class, as how to calculate it is not dependent on the geometry of the rock. The formula for density is given by

$$\rho = \frac{\Sigma_i \rho_i V_i}{\Sigma_i V_i}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **densities** (*list of floats*) – List of densities of each phase in the composite $[kg/m^3]$.

Returns
Density ρ $[kg/m^3]$.

Return type
float

average_heat_capacity_p(*fractions*, *c_p*)

Averages the heat capacities at constant pressure C_P by molar fractions.

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_p** (*list of floats*) – List of heat capacities at constant pressure C_P of each phase in the composite $[J/K/mol]$.

Returns
Heat capacity at constant pressure C_P of the composite $[J/K/mol]$.

Return type
float

average_heat_capacity_v(*fractions*, *c_v*)

Averages the heat capacities at constant volume C_V by molar fractions as in eqn. (16) in [IS92].

Parameters

- **fractions** (*list of floats*) – List of molar fractions of each phase in the composite (should sum to 1.0).
- **c_v** (*list of floats*) – List of heat capacities at constant volume C_V of each phase in the composite. $[J/K/mol]$

Returns
Heat capacity at constant volume of the composite C_V $[J/K/mol]$.

Return type
float

average_thermal_expansivity(*volumes*, *alphas*)

Averages the thermal expansion coefficient of the mineral α $[1/K]$.

Parameters

- **volumes** (*list of floats*) – List of volume fractions of each phase in the composite (should sum to 1.0).
- **alphas** (*list of floats*) – List of thermal expansivities α of each phase in the composite. $[1/K]$

Returns
Thermal expansivity of the composite α. $[1/K]$

Return type
float

6.8 Geotherms

6.9 Layers and Planets

6.9.1 Layer

class burnman.**Layer**(*name=None*, *radii=None*, *verbose=False*)

Bases: object

The base class for a planetary layer. The user needs to set the following before properties can be computed:

- set_material(), which sets the material of the layer, e.g. a mineral, solid_solution, or composite
- set_temperature_mode(), either predefine, or set to an adiabatic profile
- set_pressure_mode(), to set the self-consistent pressure (with user-defined option the pressures can be overwritten). To set the self-consistent pressure the pressure at the top and the gravity at the bottom of the layer need to be set.
- make(), computes the self-consistent part of the layer and starts the settings to compute properties within the layer

Note that the entire planet this layer sits in is not necessarily self-consistent, as the pressure at the top of the layer is a function of the density within the layer (through the gravity). Entire planets can be computed self-consistently with the planet class. Properties will be returned at the pre-defined radius array, although the evaluate() function can take a newly defined depthlist and values are interpolated between these (sufficient sampling of the layer is needed for this to be accurate).

reset()

Resets all cached material properties. It is typically not required for the user to call this function.

set_material(*material*)

Set the material of a Layer with a Material

set_temperature_mode(*temperature_mode='adiabatic'*, *temperatures=None*, *temperature_top=None*)

Sets temperatures within the layer as user-defined values or as a (potentially perturbed) adiabat.

Parameters

- **temperature_mode** (*string*) – This can be set to 'user-defined', 'adiabatic', or 'perturbed-adiabatic'. 'user-defined' fixes the temperature with the profile input by the user. 'adiabatic' self-consistently computes the adiabat when setting the state of the layer. 'perturbed-adiabatic' adds the user input array to the adiabat. This allows the user to apply boundary layers (for example).
- **temperatures** (*array of float*) – The desired fixed temperatures in [K]. Should have same length as defined radii in layer.
- **temperature_top** (*float*) – Temperature at the top of the layer. Used if the temperature mode is chosen to be 'adiabatic' or 'perturbed-adiabatic'. If 'perturbed-adiabatic' is chosen as the temperature mode, temperature_top corresponds to the true temperature at the top of the layer, and the reference isentrope at this radius is defined to lie at a temperature of temperature_top - temperatures[-1].

set_pressure_mode(*pressure_mode='self-consistent'*, *pressures=None*, *gravity_bottom=None*, *pressure_top=None*, *n_max_iterations=50*, *max_delta=1e-05*)

Sets the pressure mode of the layer, which can either be 'user-defined', or 'self-consistent'.

Parameters

- **pressure_mode** (*string*) – This can be set to 'user-defined' or 'self-consistent'. 'user-defined' fixes the pressures with the profile input by the user in the 'pressures' argument. 'self-consistent' forces Layer to calculate pressures self-consistently. If this is selected, the user will need to supply values for the gravity_bottom [m/s^2] and pressure_top [Pa] arguments.
- **pressures** (*array of floats*) – Pressures [Pa] to set layer to (if the 'user-defined' pressure_mode has been selected). The array should be the same length as the layers user-defined radii array.
- **pressure_top** (*float*) – Pressure [Pa] at the top of the layer.
- **gravity_bottom** (*float*) – Gravity [m/s^2] at the bottom of the layer.
- **n_max_iterations** (*integer*) – Maximum number of iterations to reach self-consistent pressures.
- **max_delta** (*float*) – Relative update to the highest pressure in the layer between iterations to stop iterations.

make()

This routine needs to be called before evaluating any properties. If pressures and temperatures are not user-defined, they are computed here. This method also initializes an array of copied materials from which properties can be computed.

evaluate(*properties*, *radlist=None*, *radius_planet=None*)

Function that is used to evaluate properties across the layer. If radlist is not defined, values are returned at the internal radlist. If asking for different radii than the internal radlist, pressure and temperature values are interpolated and the layer material evaluated at those pressures and temperatures.

Parameters

- **properties** (*list of strings*) – List of properties to evaluate.
- **radlist** (*array of floats*) – Radii to evaluate properties at. If left empty, internal radii list is used.
- **planet_radius** (*float*) – Planet outer radius. Used only to calculate depth.

Returns

1D or 2D array of requested properties (1D if only one property was requested)

Return type

numpy.array

property mass

Calculates the mass of the layer [kg]

property moment_of_inertia

Returns the moment of inertia of the layer [kg m^2]

property gravity

Returns gravity profile of the layer [m s^(-2)]

property bullen

Returns the Bullen parameter across the layer. The Bullen parameter assess if compression as a function of pressure is like homogeneous, adiabatic compression. Bullen parameter =1 , homogeneous, adiabatic compression Bullen parameter > 1 , more compressed with pressure, e.g. across phase transitions Bullen parameter < 1, less compressed with pressure, e.g. across a boundary layer.

property brunt_vasala

Returns the brunt-vasala (or buoyancy) frequency, N, across the layer. This frequency assess the stabilty of the layer: N < 0, fluid will convect N= 0, fluid is neutral N > 0, fluid is stabily stratified.

property pressure

Returns current pressures across the layer that was set with *set_state()*.

Aliased with *P()*.

Returns

Pressures in [Pa] at the predefined radii.

Return type

numpy.array

property temperature

Returns current temperature across the layer that was set with *set_state()*.

- Aliased with *T()*.

Returns
Temperatures in [K] at the predefined radii.

Return type
numpy.array

property molar_internal_energy

Returns the molar internal energies across the layer.

Notes

- Needs to be implemented in derived classes.
- Aliased with *energy()*.

Returns
The internal energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_gibbs

Returns the molar Gibbs free energies across the layer.

Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_helmholtz

Returns the molar Helmholtz free energies across the layer.

Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_mass

Returns molar mass of the layer.

Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
numpy.array

`property molar_volume`

Returns molar volumes across the layer.

Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volumes in [m^3/mol] at the predefined radii.

Return type
numpy.array

`property density`

Returns the densities across this layer.

Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The densities of this material in [kg/m^3] at the predefined radii.

Return type
numpy.array

`property molar_entropy`

Returns molar entropies acroos the layer.

Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropies in [J/K/mol] at the predefined radii.

Return type
numpy.array

`property molar_enthalpy`

Returns molar enthalpies across the layer.

Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpies in [J/mol] at the predefined radii.

Return type
numpy.array

`property isothermal_bulk_modulus`

Returns isothermal bulk moduli across the layer.

Notes

- Needs to be implemented in derived classes.
- Aliased with *K_T()*.

Returns
Bulk moduli in [Pa] at the predefined radii.

Return type
numpy.array

property adiabatic_bulk_modulus

Returns the adiabatic bulk moduli across the layer.

Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa] at the predefined radii.

Return type
numpy.array

property isothermal_compressibility

Returns isothermal compressibilities across the layer (or inverse isothermal bulk moduli).

Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibilities in [1/Pa] at the predefined radii.

Return type
numpy.array

property adiabatic_compressibility

Returns adiabatic compressibilities across the layer (or inverse adiabatic bulk moduli).

Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibilities in [1/Pa] at the predefined radii.

Return type
numpy.array

property shear_modulus

Returns shear moduli across the layer.

Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear moduli in [Pa] at the predefined radii.

Return type
numpy.array

property p_wave_velocity

Returns P wave speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speeds in [m/s] at the predefined radii.

Return type
numpy.array

property bulk_sound_velocity

Returns bulk sound speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocities in [m/s] at the predefined radii.

Return type
numpy.array

property shear_wave_velocity

Returns shear wave speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speeds in [m/s] at the predefined radii.

Return type
numpy.array

property grueneisen_parameter

Returns the grueneisen parameters across the layer.

Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameters [unitless] at the predefined radii.

Return type
numpy.array

property thermal_expansivity

Returns thermal expansion coefficients across the layer.

Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivities in [1/K] at the predefined radii.

Return type
numpy.array

property molar_heat_capacity_v

Returns molar heat capacity at constant volumes across the layer.

Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Heat capacities in [J/K/mol] at the predefined radii.

Return type
numpy.array

property molar_heat_capacity_p

Returns molar_heat capacity at constant pressures across the layer.

Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Heat capacities in [J/K/mol] at the predefined radii.

Return type
numpy.array

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property energy

Alias for *molar_internal_energy()*

property helmholtz

Alias for *molar_helmholtz()*

property gibbs

Alias for *molar_gibbs()*

property V

Alias for *molar_volume()*

property rho

Alias for *density()*

property S

Alias for *molar_entropy()*

property H

Alias for *molar_enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property beta_T

Alias for *isothermal_compressibility()*

property beta_S

Alias for *adiabatic_compressibility()*

property G

Alias for *shear_modulus()*

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property gr

Alias for *grueneisen_parameter()*

property alpha

Alias for *thermal_expansivity()*

property C_v

Alias for *molar_heat_capacity_v()*

property C_p

Alias for *molar_heat_capacity_p()*

class burnman.**BoundaryLayerPerturbation**(*radius_bottom*, *radius_top*, *rayleigh_number*, *temperature_change*, *boundary_layer_ratio*)

Bases: object

A class that implements a temperature perturbation model corresponding to a simple thermal boundary layer. The model takes the following form: T = a*exp((r - r1)/(r0 - r1)*c) + b*exp((r - r0)/(r1 - r0)*c) The relationships between the input parameters and a, b and c are given below.

This model is a simpler version of that proposed by [RM81].

Parameters

- **radius_bottom** (*float*) – The radius at the bottom of the layer (r0) [m].
- **radius_top** (*float*) – The radius at the top of the layer (r1) [m].
- **rayleigh_number** (*float*) – The Rayleigh number of convection within the layer. The exponential scale factor is this number to the power of 1/4 (Ra = c^4).
- **temperature_change** (*float*) – The total difference in potential temperature across the layer [K]. temperature_change = (a + b)*exp(c).
- **boundary_layer_ratio** (*float*) – The ratio of the linear scale factors (a/b) corresponding to the thermal boundary layers at the top and bottom of the layer. A

number greater than 1 implies a larger change in temperature across the top boundary than the bottom boundary.

temperature(*radii*)

Returns the temperature at one or more radii [K].

Parameters
radii (*float or numpy.array*) – The radii at which to evaluate the temperature.

Returns
The temperatures at the requested radii.

Return type
float or numpy.array

dTdr(*radii*)

Returns the thermal gradient at one or more radii [K/m].

Parameters
radii (*float or numpy.array*) – The radii at which to evaluate the thermal gradients.

Returns
The thermal gradient at the requested radii.

Return type
float or numpy.array

set_model_thermal_gradients(*dTdr_bottom*, *dTdr_top*)

Reparameterizes the model based on the thermal gradients at the bottom and top of the model.

Parameters

- **dTdr_bottom** (*float*) – The thermal gradient at the bottom of the model [K/m]. Typically negative for a cooling planet.
- **dTdr_top** (*float*) – The thermal gradient at the top of the model [K/m]. Typically negative for a cooling planet.

6.9.2 Planet

class burnman.**Planet**(*name*, *layers*, *n_max_iterations=50*, *max_delta=1e-05*, *verbose=False*)

Bases: object

A class to build (self-consistent) Planets made out of Layers (*burnman.Layer*). By default the planet is set to be self-consistent (with zero pressure at the surface and zero gravity at the center), but this can be overwritte using the set_pressure_mode(). Pressure_modes defined in the individual layers will be ignored. If temperature modes are already set for each of the layers, when the planet is initialized, the planet will be built immediately.

reset()

Resets all cached material properties. It is typically not required for the user to call this function.

get_layer(*name*)

Returns a layer with a given name

Parameters
name (*str*) – Given name of a layer

Returns
Layer with the given name.

Return type
burnman.Layer

get_layer_by_radius(*radius*)

Returns a layer in which this radius lies

Parameters
radius (*float*) – Radius at which to evaluate the layer.

Returns
Layer in which the radius lies.

Return type
burnman.Layer

evaluate(*properties*, *radlist=None*)

Function that is generally used to evaluate properties of the different layers and stitch them together. If asking for different radii than the internal radlist, pressure and temperature values are interpolated and the layer material evaluated at those pressures and temperatures.

Parameters

- **properties** (*list of strings*) – List of properties to evaluate
- **radlist** (*array of floats*) – Radii to evaluate properties at. If left empty, internal radius lists are used.

Returns
1D or 2D array of requested properties (1D if only one property was requested)

Return type
numpy.array

set_pressure_mode(*pressure_mode='self-consistent'*, *pressures=None*, *pressure_top=0.0*, *gravity_bottom=0.0*, *n_max_iterations=50*, *max_delta=1e-05*)

Sets the pressure mode of the planet by user-defined values are in a self-consistent fashion. pressure_mode is 'user-defined' or 'self-consistent'. The default for the planet is self-consistent, with zero pressure at the surface and zero pressure at the center.

Parameters

- **pressure_mode** (*str*) – This can be set to 'user-defined' or 'self-consistent'.

- **pressures** (*array of floats*) – Pressures (Pa) to set layer to ('user-defined'). This should be the same length as defined radius array for the layer.
- **pressure_top** (*float*) – Pressure (Pa) at the top of the layer.
- **gravity_bottom** (*float*) – Gravity (m/s^2) at the bottom the layer
- **n_max_iterations** (*int*) – Maximum number of iterations to reach self-consistent pressures.

make()

This routine needs to be called before evaluating any properties. If pressures and temperatures are self-consistent, they are computed across the planet here. Also initializes an array of materials in each Layer to compute properties from.

property mass

calculates the mass of the entire planet [kg]

property average_density

calculates the average density of the entire planet [kg/m^3]

property moment_of_inertia

#Returns the moment of inertia of the planet [kg m^2]

property moment_of_inertia_factor

#Returns the moment of inertia of the planet [kg m^2]

property depth

Returns depth of the layer [m]

property gravity

Returns gravity of the layer [m s^(-2)]

property bullen

Returns the Bullen parameter

property brunt_vasala

property pressure

Returns current pressure that was set with *set_state()*.

Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
array of floats

property temperature

Returns current temperature that was set with *set_state()*.

Aliased with *T()*.

Returns
Temperature in [K].

Return type
array of floats

property molar_internal_energy

Returns the molar internal energy of the planet.

Needs to be implemented in derived classes. Aliased with *energy()*.

Returns
The internal energy in [J/mol].

Return type
array of floats

property molar_gibbs

Returns the molar Gibbs free energy of the planet.

Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs energy in [J/mol].

Return type
array of floats

property molar_helmholtz

Returns the molar Helmholtz free energy of the planet.

Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz energy in [J/mol].

Return type
array of floats

property molar_mass

Returns molar mass of the planet.

Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
array of floats

property molar_volume

Returns molar volume of the planet.

Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
array of floats

property density

Returns the density of this planet.

Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
array of floats

property molar_entropy

Returns molar entropy of the planet.

Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
array of floats

property molar_enthalpy

Returns molar enthalpy of the planet.

Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
array of floats

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the planet.

Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
array of floats

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the planet.

Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
array of floats

property isothermal_compressibility

Returns isothermal compressibility of the planet (or inverse isothermal bulk modulus).

Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
array of floats

property adiabatic_compressibility

Returns adiabatic compressibility of the planet (or inverse adiabatic bulk modulus).

Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
array of floats

property shear_modulus

Returns shear modulus of the planet.

Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
array of floats

property p_wave_velocity

Returns P wave speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
array of floats

property bulk_sound_velocity

Returns bulk sound speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
array of floats

property shear_wave_velocity

Returns shear wave speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
array of floats

property grueneisen_parameter

Returns the grueneisen parameter of the planet.

Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameters [unitless].

Return type
array of floats

property thermal_expansivity

Returns thermal expansion coefficient of the planet.

Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
array of floats

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the planet.

Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
array of floats

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the planet.

Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
array of floats

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property energy

Alias for *molar_internal_energy()*

property helmholtz

Alias for *molar_helmholtz()*

property gibbs

Alias for *molar_gibbs()*

property V

Alias for *molar_volume()*

property rho

Alias for *density()*

property S

Alias for *molar_entropy()*

property H

Alias for *molar_enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus()*

property K_S

Alias for *adiabatic_bulk_modulus()*

property beta_T

Alias for *isothermal_compressibility()*

property beta_S

Alias for *adiabatic_compressibility()*

property G

Alias for *shear_modulus()*

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property gr

Alias for *grueneisen_parameter()*

property alpha

Alias for *thermal_expansivity()*

property C_v

Alias for *molar_heat_capacity_v()*

property C_p

Alias for *molar_heat_capacity_p()*

6.10 Thermodynamics

Burnman has a number of functions and classes which deal with the thermodynamics of single phases and aggregates.

6.10.1 Lattice Vibrations

6.10.1.1 Debye model

burnman.eos.debye.**debye_fn**(*x*)

Evaluate the Debye function. Takes the parameter xi = Debye_T/T

burnman.eos.debye.**debye_fn_cheb**(*x*)

Evaluate the Debye function using a Chebyshev series expansion coupled with asymptotic solutions of the function. Shamelessly adapted from the GSL implementation of the same function (Itself adapted from Collected Algorithms from ACM). Should give the same result as debye_fn(x) to near machine-precision.

burnman.eos.debye.**thermal_energy**(*T*, *debye_T*, *n*)

calculate the thermal energy of a substance. Takes the temperature, the Debye temperature, and n, the number of atoms per molecule. Returns thermal energy in J/mol

burnman.eos.debye.**molar_heat_capacity_v**(*T*, *debye_T*, *n*)

Heat capacity at constant volume. In J/K/mol

burnman.eos.debye.**helmholtz_free_energy**(*T*, *debye_T*, *n*)

Helmholtz free energy of lattice vibrations in the Debye model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyways.

burnman.eos.debye.**entropy**(*T*, *debye_T*, *n*)

Entropy due to lattice vibrations in the Debye model [J/K].

burnman.eos.debye.**dmolar_heat_capacity_v_dT**(*T*, *debye_T*, *n*)

First temperature derivative of the heat capacity at constant volume [J/K^2/mol].

6.10.1.2 Einstein model

burnman.eos.einstein.**thermal_energy**(*T*, *einstein_T*, *n*)

calculate the thermal energy of a substance. Takes the temperature, the Einstein temperature, and n, the number of atoms per molecule. Returns thermal energy in J/mol

burnman.eos.einstein.**molar_heat_capacity_v**(*T*, *einstein_T*, *n*)

Heat capacity at constant volume. In J/K/mol

burnman.eos.einstein.**helmholtz_free_energy**(*T*, *einstein_T*, *n*)

Helmholtz free energy of lattice vibrations in the Einstein model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyway.

burnman.eos.einstein.**entropy**(*T*, *einstein_T*, *n*)

Entropy due to lattice vibrations in the Einstein model [J/K]

burnman.eos.einstein.**dmolar_heat_capacity_v_dT**(*T*, *einstein_T*, *n*)

First temperature derivative of the heat capacity at constant volume according to the Einstein model [J/K^2/mol].

6.10.2 Chemistry parsing and thermodynamics

burnman.utils.chemistry.**read_masses**()

A simple function to read a file with a two column list of elements and their masses into a dictionary

burnman.utils.chemistry.**atomic_masses = {'Ag': 0.107868, 'Al': 0.0269815, 'Ar': 0.039948, 'As': 0.0749216, 'Au': 0.196967, 'B': 0.010811, 'Ba': 0.137327, 'Be': 0.00901218, 'Bi': 0.20898, 'Br': 0.079904, 'C': 0.0120107, 'Ca': 0.040078, 'Cd': 0.112411, 'Ce': 0.140116, 'Cl': 0.035453, 'Co': 0.0589332, 'Cr': 0.0519961, 'Cs': 0.132905, 'Cu': 0.063546, 'Dy': 0.1625, 'Er': 0.167259, 'Eu': 0.151964, 'F': 0.0189984, 'Fe': 0.055845, 'Ga': 0.069723, 'Gd': 0.15725, 'Ge': 0.07264, 'H': 0.00100794, 'He': 0.0040026, 'Hf': 0.17849, 'Hg': 0.20059, 'Ho': 0.16493, 'I': 0.126904, 'In': 0.114818, 'Ir': 0.192217, 'K': 0.0390983, 'Kr': 0.083798, 'La': 0.138905, 'Li': 0.006941, 'Lu': 0.174967, 'Mg': 0.024305, 'Mn': 0.054938, 'Mo': 0.09596, 'N': 0.0140067, 'Na': 0.0229898, 'Nb': 0.0929064, 'Nd': 0.144242, 'Ne': 0.0201797, 'Ni': 0.0586934, 'O': 0.0159994, 'Os': 0.19023, 'P': 0.0309738, 'Pa': 0.231036, 'Pb': 0.2072, 'Pd': 0.10642, 'Pr': 0.140908, 'Pt': 0.195084, 'Rb': 0.0854678, 'Re': 0.186207, 'Rh': 0.102905, 'Ru': 0.10107, 'S': 0.032065, 'Sb': 0.12176, 'Sc': 0.0449559, 'Se': 0.07896, 'Si': 0.0280855, 'Sm': 0.15036, 'Sn': 0.11871, 'Sr': 0.08762, 'Ta': 0.180948, 'Tb': 0.158925, 'Te': 0.1276, 'Th': 0.232038, 'Ti': 0.047867, 'Tl': 0.204383, 'Tm': 0.168934, 'U': 0.238029, 'V': 0.0509415, 'Vc': 0.0, 'W': 0.18384, 'Xe': 0.131293, 'Y': 0.0889058, 'Yb': 0.173054, 'Zn': 0.06538, 'Zr': 0.091224}**

IUPAC_element_order provides a list of all the elements. Element order is based loosely on electronegativity, following the scheme suggested by IUPAC, except that H comes after the Group 16 elements, not before them.

burnman.utils.chemistry.**dictionarize_formula**(*formula*)

A function to read a chemical formula string and convert it into a dictionary

Parameters
formula (*str*) – Chemical formula, written in the XnYm format, where the formula has n atoms of element X and m atoms of element Y

Returns
The same chemical formula, but expressed as a dictionary.

Return type
dict

burnman.utils.chemistry.**sum_formulae**(*formulae*, *amounts=None*)

Adds together a set of formulae.

Parameters

- **formulae** (*list of dictionary or counter objects*) – List of chemical formulae.
- **amounts** (*list of floats*) – List of amounts of each formula.

Returns
The sum of the user-provided formulae

Return type
Counter object

burnman.utils.chemistry.**formula_mass**(*formula*)

A function to take a chemical formula and compute the formula mass.

Parameters
formula (*dict or Counter object*) – A chemical formula

Returns
The mass per mole of formula [kg]

Return type
float

burnman.utils.chemistry.**convert_formula**(*formula*, *to_type='mass'*, *normalize=False*)

Converts a chemical formula from one type (mass or molar) into the other. Renormalises amounts if normalize=True.

Parameters

- **formula** (*dict or Counter object*) – A chemical formula.
- **to_type** (*str*) – Conversion type, one of 'mass' or 'molar'.
- **normalize** (*bool*) – Whether or not to normalize the converted formula to 1.

Returns
The converted formula.

Return type
dict

burnman.utils.chemistry.**process_solution_chemistry**(*solution_model*)

This function parses a class instance with a "formulas" attribute containing site information, e.g.

['[Mg]3[Al]2Si3O12', '[Mg]3[Mg1/2Si1/2]2Si3O12']

It outputs the bulk composition of each endmember (removing the site information), and also a set of variables and arrays which contain the site information. These are output in a format that can easily be used to calculate activities and gibbs free energies, given molar fractions of the phases and pressure and temperature where necessary.

Parameters
solution_model – Class must have a "formulas" attribute, containing a list of chemical formulae with site information

Return type
None

Note: Nothing is returned from this function, but the solution_model object gains the following attributes:

- **solution_formulae [list of dictionaries]**
  List of endmember formulae in dictionary form.
- **n_sites [integer]**
  Number of sites in the solution. Should be the same for all endmembers.
- **sites [list of lists of strings]**
  A list of species for each site in the solution.
- **site_names [list of strings]**
  A list of species_site pairs in the solution, where each distinct site is given by a unique uppercase letter e.g. ['Mg_A', 'Fe_A', 'Al_A', 'Al_B', 'Si_B'].
- **n_occupancies [integer]**
  Sum of the number of possible species on each of the sites in the solution. Example: A binary solution [[A][B],[B][C1/2D1/2]] would have n_occupancies = 5, with two possible species on Site 1 and three on Site 2.
- **site_multiplicities [2D array of floats]**
  A 1D array for each endmember in the solution, containing the multiplicities of each site per formula unit. To simplify computations later, the multiplicities are repeated for each species on each site, so the shape of this attribute is (n_endmembers, n_site_species).
- **endmember_occupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the fraction of atoms of each species on each site.
- **endmember_noccupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the number of atoms of each species on each site per mole of endmember.

burnman.utils.chemistry.**site_occupancies_to_strings**(*site_species_names*, *site_multiplicities*, *endmember_occupancies*)

Converts a list of endmember site occupancies into a list of string representations of those occupancies.

Parameters

- **site_species_names** (*2D list of strings*) – A list of list of strings, giving the names of the species which reside on each site. List of sites, each of which contains a list of the species occupying each site.
- **site_multiplicities** (*1D or 2D numpy array of floats*) – List of floats giving the multiplicity of each site. If 2D, must have the same shape as endmember_occupancies. If 1D, must be either the same length as the number of sites, or the same length as site_species_names (with an implied repetition of the same number for each species on a given site).
- **endmember_occupancies** (*2D numpy array of floats*) – A list of site-species occupancies for each endmember. The first dimension loops over the endmembers, and the second dimension loops over the site-species occupancies for that endmember. The total number and order of occupancies must be the same as the strings in site_species_names.

Returns

A list of strings in standard burnman format. For example, [Mg]3[Al]2 would correspond to the classic two-site pyrope garnet.

Return type

list of strings

burnman.utils.chemistry.**compositional_array**(*formulae*)

Parameters

formulae (*list of dicts*) – List of chemical formulae

Returns

Array of endmember formulae and a list of elements.

Return type

2D numpy.array of floats and a list of strs

burnman.utils.chemistry.**ordered_compositional_array**(*formulae*, *elements*)

Parameters

formulae (*list of dicts*) – List of chemical formulae

:param elements : List of elements :type elements: list of strings

Returns

Array of endmember formulae

Return type

2D array of floats

burnman.utils.chemistry.**formula_to_string**(*formula*)

Parameters
formula (*dict or Counter*) – Chemical formula

Returns
A formula string, with element order as given in the list IUPAC_element_order. If one or more keys in the dictionary are not one of the elements in the periodic table, then they are added at the end of the string.

Return type
str

burnman.utils.chemistry.**sort_element_list_to_IUPAC_order**(*element_list*)

:param element_list : List of elements. :type element_list: list

Returns
List of elements sorted into IUPAC order

Return type
list

burnman.utils.chemistry.**convert_fractions**(*composite*, *phase_fractions*, *input_type*, *output_type*)

Takes a composite with a set of user defined molar, volume or mass fractions (which do not have to be the fractions currently associated with the composite) and converts the fractions to molar, mass or volume.

Conversions to and from mass require a molar mass to be defined for all phases. Conversions to and from volume require set_state to have been called for the composite.

Parameters

- **composite** (*Composite*) – Composite for which fractions are to be defined.
- **phase_fractions** (*list of floats*) – List of input phase fractions (of type input_type).
- **input_type** (*str*) – Input fraction type. One of 'molar', 'mass' or 'volume'.
- **output_type** (*str*) – Output fraction type. One of 'molar', 'mass' or 'volume'.

Returns
List of output phase fractions (of type output_type)

Return type
list of floats

burnman.utils.chemistry.**reaction_matrix_as_strings**(*reaction_matrix*, *compound_names*)

Returns a list of string representations of all the reactions in reaction_matrix.

Parameters

- **reaction_matrix** (*2D numpy array*) – Matrix of stoichiometric amounts of each compound j in reaction i.

- **compound_names** (*list of strings*) – List of compound names.

Returns
List of strings corresponding to each reaction.

Return type
list of strings

burnman.tools.chemistry.**fugacity**(*standard_material*, *assemblage*)

Calculates the fugacity of a standard material in another assemblage.

Note: set_method and set_state should already have been used on both assemblages.

Parameters

- **standard_material** – Standard material for which to calculate the fugacity. The material must have a formula as a dictionary parameter.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.

Returns
Value of the fugacity of the component with respect to the standard material.

Return type
float

burnman.tools.chemistry.**relative_fugacity**(*component_formula*, *assemblage*, *reference_assemblage*)

Calculates the fugacity of a chemical component in one assemblage relative to another one.

Note: set_method and set_state should already have been used on both assemblages.

Parameters

- **component_formula** (*dictionary*) – Chemical formula for which to compute the relative fugacity.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.
- **reference_assemblage** (*burnman.Composite*) – Reference assemblage against which to measure the fugacity.

Returns
Value of the fugacity of the component in the assemblage with respect to the reference_assemblage.

Return type
float

burnman.tools.chemistry.**equilibrium_pressure**(*minerals*, *stoichiometry*, *temperature*, *pressure_initial_guess=100000.0*)

Given a list of minerals, their reaction stoichiometries and a temperature of interest, compute the equilibrium pressure of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **temperature** (*float*) – Temperature of interest [K].
- **pressure_initial_guess** (*float*) – Initial pressure guess [Pa].

Returns
The equilibrium pressure of the reaction [Pa].

Return type
float

burnman.tools.chemistry.**equilibrium_temperature**(*minerals*, *stoichiometry*, *pressure*, *temperature_initial_guess=1000.0*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**invariant_point**(*minerals_r1*, *stoichiometry_r1*, *minerals_r2*, *stoichiometry_r2*, *pressure_temperature_initial_guess=[1000000000.0, 1000.0]*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.

- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**hugoniot**(*mineral*, *P_ref*, *T_ref*, *pressures*, *reference_mineral=None*)

Calculates the temperatures (and volumes) along a Hugoniot as a function of pressure according to the Hugoniot equation U2-U1 = 0.5*(p2 - p1)(V1 - V2) where U and V are the internal energies and volumes (mass or molar) and U = F + TS

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the Hugoniot is to be calculated.
- **P_ref** (*float*) – Reference pressure [Pa]
- **T_ref** (*float*) – Reference temperature [K]
- **pressures** (*numpy.array of floats*) – Set of pressures [Pa] for which the Hugoniot temperature and volume should be calculated.
- **reference_mineral** (*burnman.Mineral*) – Mineral which is stable at the reference conditions Provides an alternative U_0 and V_0 when the reference mineral transforms to the mineral of interest at some (unspecified) pressure.

Returns
The Hugoniot temperatures and volumes at the given pressures.

Return type
tuple of numpy.arrays

burnman.tools.chemistry.**reactions_from_stoichiometric_matrix**(*stoichiometric_matrix*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters
stoichiometric_matrix (*2D numpy array*) – An array of the stoichiometric (molar) amounts of component j in compound i.

Returns
An array of the stoichiometric (molar) amounts of compound j in reaction i.

Return type
2D numpy array

burnman.tools.chemistry.**reactions_from_formulae**(*formulae*, *compound_names*, *return_strings=True*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters

- **formulae** (*list of dictionaries or list of strings*) – List of the chemical formulae, either as strings or as a list of dictionaries of elements.
- **compound_names** (*list of strings*) – List of the compound names in the formula list.
- **return_strings** (*bool*) – Whether to return the reactions as strings or array.

Returns

Either a 2D array of the stoichiometric (molar) amounts of compound j in reaction i, or a list of strings. The parameter compound_names is only used if strings are requested.

Return type

2D numpy array or list of strings

6.11 Equilibrium Thermodynamics

burnman.tools.equilibration.**calculate_constraints**(*assemblage*, *n_free_compositional_vectors*)

This function calculates the linear inequality constraints bounding the valid parameter space for a given assemblage.

The constraints are as follows:

- Pressure and temperature must be positive
- All phase fractions must be positive
- All site-species occupancies must be positive

The constraints are stored in a vector (b) and matrix (A). The sign convention is chosen such that the constraint is satisfied if A.x + b < eps.

Parameters

assemblage (*burnman.Composite*) – The assemblage for which the constraints are calculated.

Returns

The constraints vector and matrix.

Return type

tuple

burnman.tools.equilibration.**get_parameters**(*assemblage*, *n_free_compositional_vectors=0*)

Gets the starting parameters vector (x) for the current equilibrium problem. These are:

- pressure

- temperature
- absolute amount of each phase. if a phase is a solution with >1 endmember, the following parameters are the mole fractions of the independent endmembers in the solution, except for the first endmember (as the mole fractions must sum to one).

Parameters
assemblage (*burnman.Composite*) – The assemblage to be equilibrated.

Returns
The current values of all the parameters.

Return type
numpy.array

burnman.tools.equilibration.**get_endmember_amounts**(*assemblage*)

Gets the absolute amounts of all the endmembers in the solution.

Parameters
assemblage (*burnman.Composite*) – The assemblage to be equilibrated.

Returns
The current amounts of all the endmembers.

Return type
numpy.array

burnman.tools.equilibration.**set_compositions_and_state_from_parameters**(*assemblage*, *parameters*)

Sets the phase compositions, amounts and state of the assemblage from a list of parameter values.

Parameters

- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **parameters** (*numpy.array*) – The current parameter values.

burnman.tools.equilibration.**F**(*x*, *assemblage*, *equality_constraints*, *reduced_composition_vector*, *reduced_free_composition_vectors*)

The vector-valued function for which the root is sought. The first two vector values depend on the equality_constraints chosen. For example, if

- eq[i][0] = 'P', F[i] = P - eq[i][1]
- eq[i][0] = 'T', F[i] = T - eq[i][1]
- eq[i][0] = 'S', F[i] = entropy - eq[i][1]
- eq[i][0] = 'V', F[i] = volume - eq[i][1]
- eq[i][0] = 'PT_ellipse', F[i] = norm(([P, T] - eq[i][1][0])/eq[i][1][1]) - 1
- eq[i][0] = 'X', np.dot(eq[i][1][0], x) - eq[i][1][1]

The next set of vector values correspond to the reaction affinities. The final set of vector values correspond to the bulk composition constraints.

Parameters

- **x** (*numpy array*) – Parameter values for the equilibrium problem to be solved.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of lists*) – A list of the equality constraints (see above for valid formats).
- **reduced_composition_vector** (*numpy.array*) – The amounts of the independent elements.
- **reduced_free_composition_vectors** (*2D numpy.array*) – The amounts of the independent elements in each of the free_compositional_vectors.

Returns
The vector corresponding to F(x).

Return type
numpy.array

burnman.tools.equilibration.**jacobian**(*x*, *assemblage*, *equality_constraints*, *reduced_free_composition_vectors*)

The Jacobian of the vector-valued function F for which the root is sought ($\partial F/\partial x$). See documentation for *F()* and *get_parameters()* (which return F and x respectively) for more details.

Parameters

- **x** (*numpy.array*) – Parameter values for the equilibrium problem to be solved.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of lists*) – A list of the equality constraints (see documentation for burnman.tools.equilbration.F()).
- **reduced_free_composition_vectors** (*2D numpy array*) – The amounts of the independent elements in each of the free_compositional_vectors.

Returns
The Jacobian for the equilibrium problem.

Return type
2D numpy.array

burnman.tools.equilibration.**lambda_bounds**(*dx*, *x*, *endmembers_per_phase*)

Returns the lambda bounds for the damped affine invariant modification to Newton's method for non-linear problems (Deuflhard, 1974;1975;2004).

Parameters

- **dx** (*numpy.array*) – The proposed newton step.
- **x** (*numpy.array*) – Parameter values for the equilibrium problem to be solved.
- **endmembers_per_phase** (*list of int*) – A list of the number of endmembers in each phase.

Returns
Minimum and maximum allowed fractions of the full newton step (dx).

Return type
tuple of floats

burnman.tools.equilibration.**phase_fraction_constraints**(*phase*, *assemblage*, *fractions*, *prm*)

Converts a phase fraction constraint into standard linear form that can be processed by the root finding problem.

We start with a single fraction or an array of fractions for a particular phase ($n_p/\sum n = f$). These are then converted into the "X" form of constraint by multiplying by $\sum n$ and moving all terms to the LHS of the equation:

$-fn_0 - fn_1 - \ldots + (1-f)n_p - \ldots = 0$

This form is less readable, but easier to use as a constraint in a nonlinear solve.

Parameters

- **phase** (*burnman.Solution* or *burnman.Mineral*) – The phase for which the fraction is to be constrained
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **fractions** (*numpy.array*) – The phase fractions to be satified at equilibrium.
- **prm** (*namedtuple*) – A tuple with attributes n_parameters (the number of parameters for the current equilibrium problem) and phase_amount_indices (the indices of the parameters that correspond to phase amounts).

Returns
The phase fraction constraints.

Return type
list

burnman.tools.equilibration.**phase_composition_constraints**(*phase*, *assemblage*, *constraints*, *prm*)

Converts a phase composition constraint into standard linear form that can be processed by the root finding problem.

We start with constraints in the form (site_names, n, d, v), where $(nx)/(dx) = v$ and n and d are fixed vectors of site coefficients. So, one could for example choose a constraint ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) which would correspond to equal amounts Mg and Fe on the A site.

This function converts the user-defined vectors of site constraints n and d into vectors of endmember proportion constraints n' and d', such that $(n'x)/(d'x) = v$. This is done via linear transformation using the site occupancy matrix provided by *burnman.Solution*. By multiplying by the denominator, we have the following scalar comparison: $(n'x) = v(d'x)$

The equilibration function does not use the proportion of the first endmember (as the endmember proportions must sum to one), and so we split x, n' and d' into the first element and following elements: $(n'_0 x_0 + n'_i x_i) = v(d'_0 x_0 + d'_i x_i)$ where i is taken over all elements apart from the first.

With some more rearranging we can express the constraint in standard linear form: $(n'_0(1-\sum_j x_j)+n'_i x_i) = v(d'_0(1-\sum_j x_j)+d'_i x_i)$

$(n'_0+(n'_i-1_i n'_0)x_i) = v(d'_0+(d'_i-1_i d'_0)x_i)$

$(((n'_i-1_i n'_0)-v(d'_i-1_i d'_0))x_i) = (vd'_0-n'_0)$

This form is less readable, but easier to use as a constraint in a nonlinear solve.

Parameters

- **phase** (*burnman.Solution*) – The phase for which the composition is to be constrained.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **constraints** (*tuple*) – The desired constraints in the form: site_names (list of strings), numerator (numpy.array), denominator (numpy.array), values (numpy.array).

Returns

The phase composition constraints in standard form.

Return type

list

burnman.tools.equilibration.**get_equilibration_parameters**(*assemblage*, *composition*, *free_compositional_vectors*)

Builds a named tuple containing the parameter names and various other parameters needed by the equilibrium solve.

Parameters

- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **composition** (*dict*) – The bulk composition for the equilibrium problem.
- **free_compositional_vectors** (*list of dictionaries*) – The bulk compositional degrees of freedom for the equilibrium problem.

Returns

A tuple with attributes n_parameters (the number of parameters for the current equilibrium problem) and phase_amount_indices (the indices of the parameters that correspond to phase amounts).

Return type

namedtuple

burnman.tools.equilibration.**process_eq_constraints**(*equality_constraints*, *assemblage*, *prm*)

A function that processes the equality constraints into a form that can be processed by the F and jacobian functions.

This function has two main tasks: it turns phase_fraction and phase_composition constraints into standard linear constraints in the solution parameters. It also turns vector-valued constraints into a list of scalar-valued constraints.

Parameters

- **equality_constraints** (*list*) – A list of equality constraints. For valid types of constraints, please see the documentation for burnman.equilibrate().
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **prm** (*namedtuple*) – A tuple with attributes n_parameters (the number of parameters for the current equilibrium problem) and phase_amount_indices (the indices of the parameters that correspond to phase amounts).

Returns
Equality constraints in a form that can be processed by the F and jacobian functions.

Return type
list of lists

burnman.tools.equilibration.**equilibrate**(*composition*, *assemblage*, *equality_constraints*, *free_compositional_vectors=[]*, *tol=0.001*, *store_iterates=False*, *store_assemblage=True*, *max_iterations=100.0*, *verbose=False*)

A function that finds the thermodynamic equilibrium state of an assemblage subject to given equality constraints by solving a set of nonlinear equations related to the chemical potentials and other state variables of the system.

The user chooses an assemblage (e.g. olivine, garnet and orthopyroxene) and $2 + n_c$ equality constraints, where n_c is the number of bulk compositional degrees of freedom. The equilibrate function attempts to find the remaining unknowns that satisfy those constraints.

There are a number of equality constraints implemented in burnman. These are given as a list of lists. Each constraint should have the form: [<constraint type>, <constraint>], where <constraint type> is one of 'P', 'T', 'S', 'V', 'X', 'PT_ellipse', 'phase_fraction', or 'phase_composition'. The format of the <constraint> object depends on the constraint type:

- **P: float or numpy.array of**
  pressures [Pa]
- **T: float or numpy.array of**
  temperatures [K]
- **S: float or numpy.array of**
  entropies [J/K]
- **V: float or numpy.array of**
  volumes [m:math:^3]
- **PT_ellipse: list of two floats or numpy.arrays, where the equality**
  satifies the equation norm(([P, T] - arr[0])/arr[1]) = 1
- **phase_fraction: tuple with the form (<phase> <fraction(s)>),**
  where <phase> is one of the phase objects in the assemblage and <fraction(s)> is a float or numpy.array corresponding to the desired phase fractions.
- **phase_composition: tuple with the form (<phase> <constraint>),**
  where <phase> is one of the phase objects in the assemblage and <constraint> has the form (site_names, n, d, v), where $(nx)/(dx) = v$, n and d are constant vectors of site coefficients,

and v is a float or numpy.array. For example, a constraint of the form ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) would correspond to equal amounts Mg and Fe on the A site (Mg_A / (Mg_A + Fe_A) = 0.5).

- **X: list of a numpy.array and a float or numpy.array,**
  where the equality satisfies the linear equation arr[0] x = arr[1]. The first numpy.array is fixed, and the second is to be looped over by the equilibrate function. This is a generic compositional equality constraint.

Parameters

- **composition** (*dict*) – The bulk composition that the assemblage must satisfy.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of list*) – The list of equality constraints. See above for valid formats.
- **free_compositional_vectors** (*list of dict*) – A list of dictionaries containing the compositional freedom of the solution. For example, if the list contains the vector {'Mg': 1., 'Fe': -1}, that implies that the bulk composition is equal to composition + a (n_Mg - n_Fe), where the value of a is to be determined by the solve. Vector given in atomic (molar) units of elements.
- **tol** (*float*) – The tolerance for the nonlinear solver.
- **store_iterates** (*bool*) – Whether to store the parameter values for each iteration in each solution object.
- **store_assemblage** (*bool*) – Whether to store a copy of the assemblage object in each solution object.
- **max_iterations** (*int*) – The maximum number of iterations for the nonlinear solver.
- **verbose** (*bool*) – Whether to print output updating the user on the status of equilibration.

Returns

Solver solution object (or a list, or a 2D list of solution objects) created by *burnman.optimize.nonlinear_solvers.damped_newton_solve()*, and a namedtuple object created by *burnman.tools.equilibration.get_equilibration_parameters()*. See documentation of these functions for the return types. If store_assemblage is True, each solution object also has an attribute called *assemblage*, which contains a copy of the input assemblage with the equilibrated properties. So, for a 2D grid of solution objects, one could call either sols[0][1].x[0] or sols[0][1].assemblage.pressure to get the pressure.

Return type

tuple

6.12 Seismic

6.12.1 Base class for all seismic models

class burnman.classes.seismic.**Seismic1DModel**

Bases: object

Base class for all the seismological models.

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

internal_depth_list(*mindepth=0.0*, *maxdepth=1e+99*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`v_p`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_s`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_phi`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`density`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

`G`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`K`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

`QK`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

`QG`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

`depth`(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`gravity`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

6.12.2 Class for 1D Models

class burnman.classes.seismic.**SeismicTable**

Bases: *Seismic1DModel*

This is a base class that gets a 1D seismic model from a table indexed and sorted by radius. Fill the tables in the constructor after deriving from this class. This class uses `burnman.seismic.Seismic1DModel`

Note: all tables need to be sorted by increasing depth. self.table_depth needs to be defined. Alternatively, you can also overwrite the _lookup function if you want to access with something else.

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

radius(*pressure*)

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

6.12.3 Models currently implemented

class burnman.classes.seismic.**PREM**

Bases: *SeismicTable*

Reads PREM (1s) (input_seismic/prem.txt, [DA81]). See also burnman.seismic.SeismicTable.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

`QG`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

`QK`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`density`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

`depth`(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`evaluate`(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**Slow**

Bases: *SeismicTable*

Inserts the mean profiles for slower regions in the lower mantle (Lekic et al. 2012). We stitch together tables 'input_seismic/prem_lowermantle.txt', 'input_seismic/swave_slow.txt', 'input_seismic/pwave_slow.txt'). See also burnman.seismic.SeismicTable.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`evaluate`(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

`gravity`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

`pressure`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**Fast**

Bases: *SeismicTable*

Inserts the mean profiles for faster regions in the lower mantle (Lekic et al. 2012). We stitch together tables 'input_seismic/prem_lowermantle.txt', 'input_seismic/swave_fast.txt', 'input_seismic/pwave_fast.txt'). See also `burnman.seismic.Seismic1DModel`.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].

- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**STW105**

Bases: *SeismicTable*

Reads STW05 (a.k.a. REF) (1s) (input_seismic/STW105.txt, [KED08]). See also burnman.seismic.SeismicTable.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_s`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`class burnman.classes.seismic.IASP91`

Bases: *SeismicTable*

Reads REF/STW05 (input_seismic/STW105.txt, [KED08]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`K`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

`QG`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_phi`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_s`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`class burnman.classes.seismic.AK135`

Bases: *SeismicTable*

Reads AK135 (input_seismic/ak135.txt, [KEB95]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`K`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.

- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

6.12.4 Attenuation Correction

burnman.classes.seismic.**attenuation_correction**(*v_p*, *v_s*, *v_phi*, *Qs*, *Qphi*)

Applies the attenuation correction following Matas et al. (2007), page 4. This is simplified, and there is also currently no 1D Q model implemented. The correction, however, only slightly reduces the velocities, and can be ignored for our current applications. Arguably, it might not be as relevant when comparing computations to PREM for periods of 1s as is implemented here. Called from burnman.main.apply_attenuation_correction()

Parameters

- **v_p** (*float*) – P wave velocity in [m/s].
- **v_s** (*float*) – S wave velocitiy in [m/s].
- **v_phi** (*float*) – Bulk sound velocity in [m/s].
- **Qs** (*float*) – Shear quality factor [dimensionless].

- **Qphi** (*float*) – Bulk quality factor [dimensionless].

Returns
Corrected P wave, S wave and bulk sound velocities in [m/s].

Return type
tuple

6.13 Mineral databases

Mineral database

- *SLB_2005*
- *SLB_2011_ZSB_2013*
- *SLB_2011*
- *DKS_2013_liquids*
- *DKS_2013_solids*
- *RS_2014_liquids*
- *Murakami_etal_2012*
- *Murakami_2013*
- *Matas_etal_2007*
- *HP_2011_ds62*
- *HP_2011_fluids*
- *HHPH_2013*
- *HGP_2018_ds633*
- *SE_2015*
- *other*

6.13.1 Matas_etal_2007

Minerals from Matas et al. 2007 and references therein. See Table 1 and 2.

class burnman.minerals.Matas_etal_2007.**mg_perovskite**
Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**fe_perovskite**
Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**al_perovskite**
Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**ca_perovskite**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**periclase**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**wuestite**

Bases: *Mineral*

burnman.minerals.Matas_etal_2007.**ca_bridgmanite**

alias of *ca_perovskite*

burnman.minerals.Matas_etal_2007.**mg_bridgmanite**

alias of *mg_perovskite*

burnman.minerals.Matas_etal_2007.**fe_bridgmanite**

alias of *fe_perovskite*

burnman.minerals.Matas_etal_2007.**al_bridgmanite**

alias of *al_perovskite*

6.13.2 Murakami_etal_2012

Minerals from Murakami et al. (2012) supplementary table 5 and references therein, V_0 from Stixrude & Lithgow-Bertolloni 2005. Some information from personal communication with Murakami.

class burnman.minerals.Murakami_etal_2012.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_perovskite_3rdorder**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_periclase**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase**

Bases: *HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_3rd**

Bases: *HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS_3rd**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS_3rd**

Bases: *Mineral*

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite**

alias of *mg_perovskite*

burnman.minerals.Murakami_etal_2012.**fe_bridgmanite**

alias of *fe_perovskite*

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite_3rdorder**

alias of *mg_perovskite_3rdorder*

6.13.3 Murakami_2013

Minerals from Murakami 2013 and references therein.

class burnman.minerals.Murakami_2013.**periclase**

Bases: *Mineral*

class burnman.minerals.Murakami_2013.**wuestite**

Bases: *Mineral*

Murakami 2013 and references therein

class burnman.minerals.Murakami_2013.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_2013.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.Murakami_2013.**mg_bridgmanite**

alias of *mg_perovskite*

burnman.minerals.Murakami_2013.**fe_bridgmanite**

alias of *fe_perovskite*

6.13.4 SLB_2005

Minerals from Stixrude & Lithgow-Bertelloni 2005 and references therein

class burnman.minerals.SLB_2005.**stishovite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.SLB_2005.**mg_bridgmanite**

alias of *mg_perovskite*

burnman.minerals.SLB_2005.**fe_bridgmanite**

alias of *fe_perovskite*

6.13.5 SLB_2011

Minerals from Stixrude & Lithgow-Bertelloni 2011 and references therein. File autogenerated using SLB-data_to_burnman.py.

class burnman.minerals.SLB_2011.**c2c_pyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**ca_ferrite_structured_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**akimotoite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**post_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_ringwoodite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_aluminous_spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_wadsleyite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**anorthite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**albite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**spinel**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**hercynite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**forsterite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fayalite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_wadsleyite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_wadsleyite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_ringwoodite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_ringwoodite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**enstatite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**ferrosilite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_tschermaks**

Bases: *Mineral*

class burnman.minerals.SLB_2011.ortho_diopside

Bases: *Mineral*

class burnman.minerals.SLB_2011.diopside

Bases: *Mineral*

class burnman.minerals.SLB_2011.hedenbergite

Bases: *Mineral*

class burnman.minerals.SLB_2011.clinoenstatite

Bases: *Mineral*

class burnman.minerals.SLB_2011.ca_tschermaks

Bases: *Mineral*

class burnman.minerals.SLB_2011.jadeite

Bases: *Mineral*

class burnman.minerals.SLB_2011.hp_clinoenstatite

Bases: *Mineral*

class burnman.minerals.SLB_2011.hp_clinoferrosilite

Bases: *Mineral*

class burnman.minerals.SLB_2011.ca_perovskite

Bases: *Mineral*

class burnman.minerals.SLB_2011.mg_akimotoite

Bases: *Mineral*

class burnman.minerals.SLB_2011.fe_akimotoite

Bases: *Mineral*

class burnman.minerals.SLB_2011.corundum

Bases: *Mineral*

class burnman.minerals.SLB_2011.pyrope

Bases: *Mineral*

class burnman.minerals.SLB_2011.almandine

Bases: *Mineral*

class burnman.minerals.SLB_2011.grossular

Bases: *Mineral*

class burnman.minerals.SLB_2011.mg_majorite

Bases: *Mineral*

class burnman.minerals.SLB_2011.jd_majorite

Bases: *Mineral*

class burnman.minerals.SLB_2011.**quartz**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**coesite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**stishovite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**seifertite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**al_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**al_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**na_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**kyanite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**nepheline**

Bases: *Mineral*

burnman.minerals.SLB_2011.**ab**
alias of *albite*

burnman.minerals.SLB_2011.**an**
alias of *anorthite*

burnman.minerals.SLB_2011.**sp**
alias of *spinel*

burnman.minerals.SLB_2011.**hc**
alias of *hercynite*

burnman.minerals.SLB_2011.**fo**
alias of *forsterite*

burnman.minerals.SLB_2011.**fa**
alias of *fayalite*

burnman.minerals.SLB_2011.**mgwa**
alias of *mg_wadsleyite*

burnman.minerals.SLB_2011.**fewa**
alias of *fe_wadsleyite*

burnman.minerals.SLB_2011.**mgri**
alias of *mg_ringwoodite*

burnman.minerals.SLB_2011.**feri**
alias of *fe_ringwoodite*

burnman.minerals.SLB_2011.**en**
alias of *enstatite*

burnman.minerals.SLB_2011.**fs**
alias of *ferrosilite*

burnman.minerals.SLB_2011.**mgts**
alias of *mg_tschermaks*

burnman.minerals.SLB_2011.**odi**
alias of *ortho_diopside*

burnman.minerals.SLB_2011.**di**
alias of *diopside*

burnman.minerals.SLB_2011.**he**
alias of *hedenbergite*

burnman.minerals.SLB_2011.**cen**
alias of *clinoenstatite*

burnman.minerals.SLB_2011.**cats**
alias of *ca_tschermaks*

burnman.minerals.SLB_2011.**jd**
alias of *jadeite*

burnman.minerals.SLB_2011.**mgc2**
alias of *hp_clinoenstatite*

burnman.minerals.SLB_2011.**fec2**
alias of *hp_clinoferrosilite*

burnman.minerals.SLB_2011.**hpcen**
alias of *hp_clinoenstatite*

burnman.minerals.SLB_2011.**hpcfs**
alias of *hp_clinoferrosilite*

burnman.minerals.SLB_2011.**mgpv**
alias of *mg_perovskite*

burnman.minerals.SLB_2011.**mg_bridgmanite**
alias of *mg_perovskite*

burnman.minerals.SLB_2011.**fepv**
alias of *fe_perovskite*

burnman.minerals.SLB_2011.**fe_bridgmanite**
alias of *fe_perovskite*

burnman.minerals.SLB_2011.**alpv**
alias of *al_perovskite*

burnman.minerals.SLB_2011.**capv**
alias of *ca_perovskite*

burnman.minerals.SLB_2011.**mgil**
alias of *mg_akimotoite*

burnman.minerals.SLB_2011.**feil**
alias of *fe_akimotoite*

burnman.minerals.SLB_2011.**co**
alias of *corundum*

burnman.minerals.SLB_2011.**py**
alias of *pyrope*

burnman.minerals.SLB_2011.**al**
alias of *almandine*

burnman.minerals.SLB_2011.**gr**
alias of *grossular*

burnman.minerals.SLB_2011.**mgmj**
alias of *mg_majorite*

burnman.minerals.SLB_2011.**jdmj**
alias of *jd_majorite*

burnman.minerals.SLB_2011.**qtz**
alias of *quartz*

burnman.minerals.SLB_2011.**coes**
alias of *coesite*

burnman.minerals.SLB_2011.**st**
alias of *stishovite*

burnman.minerals.SLB_2011.**seif**
alias of *seifertite*

burnman.minerals.SLB_2011.**mppv**
alias of *mg_post_perovskite*

burnman.minerals.SLB_2011.**fppv**
alias of *fe_post_perovskite*

burnman.minerals.SLB_2011.**appv**
alias of *al_post_perovskite*

burnman.minerals.SLB_2011.**pe**
alias of *periclase*

burnman.minerals.SLB_2011.**wu**
alias of *wuestite*

burnman.minerals.SLB_2011.**mgcf**
alias of *mg_ca_ferrite*

burnman.minerals.SLB_2011.**fecf**
alias of *fe_ca_ferrite*

burnman.minerals.SLB_2011.**nacf**
alias of *na_ca_ferrite*

burnman.minerals.SLB_2011.**ky**
alias of *kyanite*

burnman.minerals.SLB_2011.**neph**
alias of *nepheline*

burnman.minerals.SLB_2011.**c2c**
alias of *c2c_pyroxene*

burnman.minerals.SLB_2011.**cf**
alias of *ca_ferrite_structured_phase*

burnman.minerals.SLB_2011.**cpx**
alias of *clinopyroxene*

burnman.minerals.SLB_2011.**gt**
alias of *garnet*

burnman.minerals.SLB_2011.**il**
alias of *akimotoite*

burnman.minerals.SLB_2011.**ilmenite_group**
alias of *akimotoite*

burnman.minerals.SLB_2011.**mw**
alias of *ferropericlase*

burnman.minerals.SLB_2011.**magnesiowuestite**
alias of *ferropericlase*

burnman.minerals.SLB_2011.**ol**
alias of *mg_fe_olivine*

burnman.minerals.SLB_2011.**opx**
alias of *orthopyroxene*

burnman.minerals.SLB_2011.**plag**
alias of *plagioclase*

burnman.minerals.SLB_2011.**ppv**
alias of *post_perovskite*

burnman.minerals.SLB_2011.**pv**
alias of *mg_fe_perovskite*

burnman.minerals.SLB_2011.**mg_fe_bridgmanite**
alias of *mg_fe_perovskite*

burnman.minerals.SLB_2011.**mg_fe_silicate_perovskite**
alias of *mg_fe_perovskite*

burnman.minerals.SLB_2011.**ri**
alias of *mg_fe_ringwoodite*

burnman.minerals.SLB_2011.**spinel_group**
alias of *mg_fe_aluminous_spinel*

burnman.minerals.SLB_2011.**wa**

alias of *mg_fe_wadsleyite*

burnman.minerals.SLB_2011.**spinelloid_III**

alias of *mg_fe_wadsleyite*

6.13.6 SLB_2011_ZSB_2013

Minerals from Stixrude & Lithgow-Bertelloni 2011, Zhang, Stixrude & Brodholt 2013, and references therein.

class burnman.minerals.SLB_2011_ZSB_2013.**stishovite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.SLB_2011_ZSB_2013.**mg_bridgmanite**

alias of *mg_perovskite*

burnman.minerals.SLB_2011_ZSB_2013.**fe_bridgmanite**

alias of *fe_perovskite*

6.13.7 DKS_2013_solids

Solids from de Koker and Stixrude (2013) FPMD simulations

class burnman.minerals.DKS_2013_solids.**stishovite**

Bases: *Mineral*

class burnman.minerals.DKS_2013_solids.**perovskite**

Bases: *Mineral*

class burnman.minerals.DKS_2013_solids.**periclase**

Bases: *Mineral*

6.13.8 DKS_2013_liquids

Liquids from de Koker and Stixrude (2013) FPMD simulations.

burnman.minerals.DKS_2013_liquids.**vector_to_array**(*a*, *Of*, *Otheta*)

class burnman.minerals.DKS_2013_liquids.**SiO2_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSiO3_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi2O5_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi3O7_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi5O11_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg2SiO4_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg3Si2O7_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg5SiO7_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgO_liquid**

Bases: *Mineral*

6.13.9 RS_2014_liquids

Liquids from Ramo and Stixrude (2014) FPMD simulations. There are some typos in the article which have been corrected where marked with the help of David Munoz Ramo.

class burnman.minerals.RS_2014_liquids.**Fe2SiO4_liquid**

Bases: *Mineral*

6.13.10 HP_2011_ds62

Endmember minerals from Holland and Powell 2011 and references therein. Update to dataset version 6.2. The values in this document are all in S.I. units, unlike those in the original tc-ds62.txt. File autogenerated using HPdata_to_burnman.py.

class burnman.minerals.HP_2011_ds62.**fo**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fa

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.teph

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.lrn

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mont

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.chum

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.chdr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mwd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fwd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mrw

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.frw

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.apv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mak

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fak

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.maj

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**py**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**alm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spss**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**gr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**andr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**knor**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osma**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osmm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osfa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**vsv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**andalusite**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ky**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sill**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**smul**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**amul**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tpz**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**merw**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**zo**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cz**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ep**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fep**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pmt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**law**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mpm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fpm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**jgd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**geh**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ak**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**rnk**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ty**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**crd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hcrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mncrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**phA**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cstn**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**zrc**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**en**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pren**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cen**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hen**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fs**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mgts**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.di

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.hed

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.jd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.acm

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.kos

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cats

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.caes

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.rhod

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pxmn

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.wo

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pswo

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.wal

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.tr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fact

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ts

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.parg

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.gl

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fgl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**rieb**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**anth**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fanth**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cumm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**grun**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ged**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spr4**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spr5**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fspr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mcar**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcar**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**deer**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cel**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcel**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ma**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**phl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ann**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnbi**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**east**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**naph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**clin**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ames**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**afchl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**daph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnchl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sud**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fsud**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**prl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ta**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fta**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tats**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tap**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**minn**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**minm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**kao**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pre**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fpre**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**chr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**liz**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**glt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fstp**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mstp**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**atg**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ab**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**abh**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mic**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**san**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**an**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**kcm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**wa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hol**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**q**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**crst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**coe**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**stv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ne**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cg**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cgh**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sdl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**kls**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lc**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**me**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**wrk**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lmt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**heu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**stlb**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**anl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lime**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ru**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**per**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fper**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mang**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cor**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mcor**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hem**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**esk**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**bix**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**NiO**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pnt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**geik**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ilm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.bdy

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ten

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cup

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.sp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.herc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mt

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mft

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.usp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.picr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.br

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.dsp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.gth

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.arag

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mag

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.sid

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.rhc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**dol**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ank**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**syv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hlt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pyr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trot**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tro**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lot**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trov**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**any**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**iron**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**Ni**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**Cu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**gph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**diam**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**S**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**syvL**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.hltL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.perL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.limL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.corL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.qL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.h2oL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.foL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.faL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.woL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.enL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.diL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.silL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.anL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.kspL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.abL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.neL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.lcL

Bases: *Mineral*

burnman.minerals.HP_2011_ds62.**cov**()

A function which loads and returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset.

Returns

cov

[dictionary] Dictionary keys are: - endmember_names: a list of endmember names, and - covariance_matrix: a 2D variance-covariance array for the

endmember zero-point energies of formation

6.13.11 HP_2011_fluids

Fluids from Holland and Powell 2011 and references therein. CORK parameters are taken from various sources.

CHO gases from Holland and Powell, 1991:

- ["CO2",304.2,0.0738]
- ["CH4",190.6,0.0460]
- ["H2",41.2,0.0211]
- ["CO",132.9,0.0350]

H2O and S2 from Wikipedia, 2012/10/23:

- ["H2O",647.096,0.22060]
- ["S2",1314.00,0.21000]

H2S from ancyclopedia.airliquide.com, 2012/10/23:

- ["H2S",373.15,0.08937]

NB: Units for cork[i] in Holland and Powell datasets are:

- a = kJ^2/kbar*K^(1/2)/mol^2: multiply by 1e-2
- b = kJ/kbar/mol: multiply by 1e-5
- c = kJ/kbar^1.5/mol: multiply by 1e-9
- d = kJ/kbar^2/mol: multiply by 1e-13

Individual terms are divided through by P, P, P^1.5, P^2, so:

- [0][j]: multiply by 1e6
- [1][j]: multiply by 1e3
- [2][j]: multiply by 1e3
- [3][j]: multiply by 1e3
- cork_P is given in kbar: multiply by 1e8

class burnman.minerals.HP_2011_fluids.**CO2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**CH4**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**O2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**H2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**S2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**H2S**

Bases: *Mineral*

6.13.12 HHPH_2013

Minerals from Holland et al. (2013) and references therein. The values in this document are all in S.I. units, unlike those in the original paper. File autogenerated using HHPHdata_to_burnman.py.

class burnman.minerals.HHPH_2013.**fo**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fa**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mwd**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fwd**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mrw**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**frw**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**apv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**npv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mak**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fak**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**maj**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**nagt**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**py**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**alm**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**gr**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**en**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cen**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hen**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hfs**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fs**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mgts**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**di**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hed**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.jd

Bases: *Mineral*

class burnman.minerals.HHPH_2013.cats

Bases: *Mineral*

class burnman.minerals.HHPH_2013.stv

Bases: *Mineral*

class burnman.minerals.HHPH_2013.macf

Bases: *Mineral*

class burnman.minerals.HHPH_2013.mscf

Bases: *Mineral*

class burnman.minerals.HHPH_2013.fscf

Bases: *Mineral*

class burnman.minerals.HHPH_2013.nacf

Bases: *Mineral*

class burnman.minerals.HHPH_2013.cacf

Bases: *Mineral*

class burnman.minerals.HHPH_2013.manal

Bases: *Mineral*

class burnman.minerals.HHPH_2013.nanal

Bases: *Mineral*

class burnman.minerals.HHPH_2013.msnal

Bases: *Mineral*

class burnman.minerals.HHPH_2013.fsnal

Bases: *Mineral*

class burnman.minerals.HHPH_2013.canal

Bases: *Mineral*

class burnman.minerals.HHPH_2013.per

Bases: *Mineral*

class burnman.minerals.HHPH_2013.fper

Bases: *Mineral*

class burnman.minerals.HHPH_2013.cor

Bases: *Mineral*

class burnman.minerals.HHPH_2013.mcor

Bases: *Mineral*

6.13.13 HGP_2018_ds633

Endmember minerals and melt solutions from Holland, Green and Powell (2018) and references therein. Dataset version 6.33. The values in this document are all in S.I. units, unlike those in the original tc-ds633.txt. The endmember section of this file is autogenerated using HGP633data_to_burnman.py.

class burnman.minerals.HGP_2018_ds633.**fo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**teph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lrn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mont**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chum**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chdr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mwd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fwd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mrw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**frw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**apv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**npv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ppv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**maj**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nagt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**py**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**alm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spss**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**andr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ski**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**knor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**uv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osma**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osmm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osfa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**vsv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**andalusite**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ky**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sill**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**smul**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**amul**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tpz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**merw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ep**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fep**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pmt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**law**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mpm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**jgd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**geh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rnk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ty**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**crd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hcrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mncrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phA**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phD**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phE**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**shB**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cstn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zrc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zrt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tcn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**en**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pren**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cen**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hen**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hfs**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fs**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mgts**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**di**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hed**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**jd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kjd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**acm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kos**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cats**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**caes**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rhod**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pxmn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pswo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fact**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ts**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**parg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fgl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nyb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rieb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fanth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cumm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**grun**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ged**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spr4**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spr5**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fspr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mcar**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcar**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**deer**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cel**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcel**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ma**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ann**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnbi**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**east**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**naph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tan**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**clin**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ames**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**afchl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**daph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnchl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sud**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fsud**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**prl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tats**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tap**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**minn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**minm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kao**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pre**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpre**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**liz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**glt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fstp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mstp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**atg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ab**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**abh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mic**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**san**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**an**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kcm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hol**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**q**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**crst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**coe**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**stv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ne**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cgh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**macf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mscf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fscf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nacf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cacf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**manal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nanal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**msnal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fsnal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**canal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sdl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kls**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**me**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wrk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lmt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**heu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**stlb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lime**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ru**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**per**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fper**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mang**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mcor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hem**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**esk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bix**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**NiO**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pnt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**geik**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ilm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdy**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyT**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyC**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ten**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cup**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**herc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mft**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**qnd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**usp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**picr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**br**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**dsp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**arag**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mag**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sid**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rhc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**dol**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ank**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**syv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hlt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pyr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trot**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tro**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lot**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trov**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**any**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**iron**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**Ni**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**Cu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**diam**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**S**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**syvL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hltL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**perL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**limL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**corL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**eskL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hemL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**qL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**h2oL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**foL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**faL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**woL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**enL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**diL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**silL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kspL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**abL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**neL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lcL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ruL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyL**

Bases: *Mineral*

burnman.minerals.HGP_2018_ds633.**cov**()

A function which loads and returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset.

Returns

cov
[dictionary] Dictionary keys are: - endmember_names: a list of endmember names, and - covariance_matrix: a 2D variance-covariance array for the

endmember zero-point energies of formation

burnman.minerals.HGP_2018_ds633.**make_melt_class**(*selected_endmembers*)

A function that generates a melt class that is a subclass of the Holland et al. (2018) silicate melt model.

Parameters

selected_endmembers: list of Minerals
Endmembers to include in the model. Valid endmembers are given in the following list, and should be specified in the same order as they appear in the list: ['q4L', 'sl1L', 'wo1L', 'fo2L', 'fa2L', 'jdL', 'hmL', 'ekL', 'tiL', 'kjL', 'ctL', 'h2o1L'].

Returns

melt_class: Solution class
Melt class spanning the specified endmembers.

class burnman.minerals.HGP_2018_ds633.**silicate_melt**(*molar_fractions=None*)

Bases: *Solution*

burnman.minerals.HGP_2018_ds633.**CMS_melt**

alias of silicate_melt

burnman.minerals.HGP_2018_ds633.**MS_melt**

alias of silicate_melt

6.13.14 JH_2015

Solid solutions from Jennings and Holland, 2015 and references therein (10.1093/petrology/egv020). The values in this document are all in S.I. units, unlike those in the original tc file.

class burnman.minerals.JH_2015.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**cfs**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**crdi**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**cess**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**cen**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**cfm**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**fm**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**odi**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**cren**

Bases: CombinedMineral

class burnman.minerals.JH_2015.**mess**

Bases: CombinedMineral

burnman.minerals.JH_2015.**construct_combined_covariance**(*original_covariance_dictionary*, *combined_mineral_list*)

This function takes a dictionary containing a list of endmember_names and a covariance_matrix, and a list of CombinedMineral instances, and creates an updated covariance dictionary containing those CombinedMinerals

Parameters

original_covariance_dictionary

[dictionary] Contains a list of strings of endmember_names of length n and a 2D numpy array covariance_matrix of shape n x n

combined_mineral_list
[list of instances of `burnman.CombinedMineral`] List of minerals to be added to the covariance matrix

Returns

cov
[dictionary] Updated covariance dictionary, with the same keys as the original

`burnman.minerals.JH_2015.cov()`

A function which returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset. Derived from HP_2011_ds62, modified to include all the new CombinedMinerals.

Returns

cov
[dictionary] Dictionary keys are: - endmember_names: a list of endmember names, and - covariance_matrix: a 2D variance-covariance array for the endmember enthalpies of formation

6.13.15 Saxena and Eriksson (2015)

Iron endmember minerals and melt taken from [SE15] using the equation of state of [BMS07].

1 bar gibbs free energy coefficients are given in the following order: [[T_max, [const, T, T*ln(T), T^(-1), T^(-2), T^(-3), T^(-9), T^2, T^3, T^4, T^7, T^(1/2), ln(T)]]

`class burnman.minerals.SE_2015.bcc_iron`

Bases: *Mineral*

BCC iron from [SE15].

`class burnman.minerals.SE_2015.fcc_iron`

Bases: *Mineral*

FCC iron from [SE15].

`class burnman.minerals.SE_2015.hcp_iron`

Bases: *Mineral*

HCP iron from [SE15].

`class burnman.minerals.SE_2015.liquid_iron`

Bases: *Mineral*

Liquid iron from [SE15].

6.13.16 Other minerals

class burnman.minerals.other.**liquid_iron**

Bases: *Mineral*

Liquid iron equation of state from Anderson and Ahrens (1994)

class burnman.minerals.other.**ZSB_2013_mg_perovskite**

Bases: *Mineral*

class burnman.minerals.other.**ZSB_2013_fe_perovskite**

Bases: *Mineral*

class burnman.minerals.other.**Speziale_fe_periclase**

Bases: *HelperSpinTransition*

class burnman.minerals.other.**Speziale_fe_periclase_HS**

Bases: *Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Speziale_fe_periclase_LS**

Bases: *Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Liquid_Fe_Anderson**

Bases: *Mineral*

Anderson & Ahrens, 1994 JGR

class burnman.minerals.other.**Fe_Dewaele**

Bases: *Mineral*

Dewaele et al., 2006, Physical Review Letters

6.14 Calibrant databases

Calibrant database

- *Decker_1971*

6.14.1 Decker_1971

class burnman.calibrants.Decker_1971.**NaCl_B1**

Bases: *Calibrant*

The NaCl (B1 structured) pressure standard reported by Decker (1971).

Note: This calibrant is not exactly the same as that proposed by Decker. The cold compression curve has here been approximated by a Birch-Murnaghan EoS.

6.15 Optimization

burnman.optimize.composition_fitting.**fit_composition_to_solution**(*solution*, *fitted_variables*, *variable_values*, *variable_covariances*, *variable_conversions=None*, *normalize=True*)

Takes a Solution object and a set of variable names and associates values and covariances and finds the molar fractions of the solution which provide the best fit (in a least-squares sense) to the variable values.

The fitting applies appropriate non-negativity constraints (i.e. no species can have a negative occupancy on a site).

Parameters

- **solution** (*burnman.Solution*) – The solution to use in the fitting procedure.
- **fitted_variables** (*list of str*) – A list of the variables used to find the best-fit molar fractions of the solution. These should either be elements such as "Fe", site_species such as "Fef_B" which would correspond to a species labelled Fef on the second site, or user-defined variables which are arithmetic sums of elements and/or site_species defined in "variable_conversions".
- **variable_values** (*numpy.array*) – Numerical values of the fitted variables. These should be given as amounts; they do not need to be normalized.
- **variable_covariances** (*2D numpy.array*) – Covariance matrix of the variables.
- **variable_conversions** (*dict of dict, or None*) – A dictionary converting any user-defined variables into an arithmetic sum of element and site-species amounts. For example, {'Mg_equal': {'Mg_A': 1., 'Mg_B': -1.}}, coupled with Mg_equal = 0 would impose a constraint that the amount of Mg would be equal on the first and second site in the solution.
- **normalize** (*bool*) – If True, normalizes the optimized molar fractions to sum to unity.

Returns

Optimized molar fractions, corresponding covariance matrix and the weighted residual.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

burnman.optimize.composition_fitting.**fit_phase_proportions_to_bulk_composition**(*phase_compositions*, *bulk_composition*)

Performs weighted constrained least squares on a set of phase compositions to find the amount of those phases that best-fits a given bulk composition.

The fitting applies appropriate non-negativity constraints (i.e. no phase can have a negative abundance in the bulk).

Parameters

- **phase_compositions** (*2D numpy.array*) – The composition of each phase. Can be in weight or mole amounts.
- **bulk_composition** (*numpy.array*) – The bulk composition of the composite. Must be in the same units as the phase compositions.

Returns
Optimized molar fractions, corresponding covariance matrix and the weighted residual.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

class burnman.optimize.eos_fitting.**MineralFit**(*mineral*, *data*, *data_covariances*, *flags*, *fit_params*, *mle_tolerances*, *delta_params=None*, *bounds=None*)

Bases: object

Class for fitting mineral parameters to experimental data. Instances of this class are passed to burnman.nonlinear_least_squares_fit().

For attributes added to this model when fitting is done, please see the documentation for that function.

set_params(*param_values*)

get_params()

function(*x*, *flag*)

normal(*x*, *flag*)

burnman.optimize.eos_fitting.**fit_PTp_data**(*mineral*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=50*, *verbose=True*)

Given a mineral of any type, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the parameters should be optimized.
- **fit_params** (*list of str*) – List of dictionary keys contained in mineral.params corresponding to the variables to be optimized during fitting. Initial guesses are taken from the existing values for the parameters
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional) If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns

Model with optimized parameters.

Return type

burnman.optimize.eos_fitting.MineralFit

burnman.optimize.eos_fitting.**fit_PTV_data**(*mineral*, *fit_params*, *data*, *data_covariances=[]*, *delta_params=None*, *bounds=None*, *param_tolerance=1e-05*, *max_lm_iterations=50*, *verbose=True*)

A simple alias for the fit_PTp_data for when all the data is volume data

class burnman.optimize.eos_fitting.**SolutionFit**(*solution*, *data*, *data_covariances*, *flags*, *fit_params*, *mle_tolerances*, *delta_params=None*, *bounds=None*)

Bases: object

Class for fitting mineral parameters to experimental data. Instances of this class are passed to burnman.nonlinear_least_squares_fit().

For attributes added to this model when fitting is done, please see the documentation for that function.

set_params(*param_values*)

get_params()

function(*x*, *flag*)

normal(*x*, *flag*)

burnman.optimize.eos_fitting.**fit_XPTp_data**(*solution*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=50*, *verbose=True*)

Given a symmetric solution, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **solution** (*burnman.Solution*) – Solution for which the parameters should be optimized.
- **fit_params** (*list of lists*) – Variables to be optimized during fitting. Each list is either of length two or three. The first item of length-2 lists should be a dictionary key contained in one of the endmember mineral.params, and the second item should be the index of the endmember in the solution (indexing starts from 0). The first item of length-3 lists should be one of 'E', 'S' or 'V' (the excess energies, entropies or volumes in each binary). The second two items should be the indices of the pair of endmembers bounding the binary, in ascending order (indexing starts from 0). Initial guesses are taken from the existing values for the parameters.
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional). If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).

- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns
Model with optimized parameters.

Return type
burnman.optimize.eos_fitting.SolutionFit

burnman.optimize.linear_fitting.**weighted_constrained_least_squares**(*A*, *b*, *Cov_b=None*, *equality_constraints=None*, *inequality_constraints=None*)

Solves a weighted, constrained least squares problem using cvxpy. The objective function is to minimize the following: sum_squares(Cov_b^(-1/2).A.x - Cov_b^(-1/2).b)) subject to C.x == c D.x <= d

Parameters

- **A** (*2D numpy array*) – An array defining matrix A in the objective function above.
- **b** (*numpy array*) – An array defining vector b in the objective function above.
- **Cov_b** (*2D numpy array*) – A covariance matrix associated with b
- **equality_constraints** (*list containing a 2D array and 1D array*) – A list containing the matrices C and c in the objective function above.
- **inequality_constraints** (*list containing a 2D array and 1D array*) – A list containing the matrices D and d in the objective function above.

Returns
Tuple containing the optimized phase amounts (1D numpy.array), a covariance matrix corresponding to the optimized phase amounts (2D numpy.array), and the weighted residual of the fitting procedure (a float).

Return type
tuple

burnman.optimize.nonlinear_fitting.**nonlinear_least_squares_fit**(*model*, *lm_damping=0.0*, *param_tolerance=1e-07*, *max_lm_iterations=100*, *verbose=False*)

Function to compute the "best-fit" parameters for a model by nonlinear least squares fitting.

The nonlinear least squares algorithm closely follows the logic in Section 23.1 of Bayesian Probability Theory (von der Linden et al., 2014; Cambridge University Press).

Parameters

:param model: Model containing data to be fit, and functions to
aid in fitting.

:type model: object
:param lm_damping: Levenberg-Marquardt parameter for least squares minimization.
:type lm_damping: float
:param param_tolerance: Levenberg-Marquardt iterations are terminated when
the maximum fractional change in any of the parameters during an iteration drops below this value

:type param_tolerance: float
:param max_lm_iterations: Maximum number of Levenberg-Marquardt iterations
:type max_lm_iterations: int
:param verbose: Print some information to standard output
:type verbose: bool
.. note:: The object passed as model must have the following attributes:

- **data [2D numpy.array] - Elements of x[i][j] contain the**
observed position of data point i.
- **data_covariances [3D numpy.array] Elements of cov[i][j][k] contain**
the covariance matrix of data point i.
- **mle_tolerances [numpy.array] - The iterations to find the maximum likelihood**
estimator for each observed data point will stop when mle_tolerances[i] < np.linalg.norm(data_mle[i] - model.function(data_mle[i], flag))
- delta_params [numpy.array] - parameter perturbations used to compute the jacobian

Must also have the following methods: * set_params(self, param_values) - Function to set parameters. * get_params(self) - Function to get current model parameters. * function(self, x) - Returns value of model function evaluated at x. * normal(self, x) - Returns value of normal to the model function evaluated at x.

After this function has been performed, the following attributes are added to model:

- n_dof [int] - Degrees of freedom of the system.
- **data_mle [2D numpy array] - Maximum likelihood estimates of the observed data points**
on the best-fit curve.
- jacobian [2D numpy array] - d(weighted_residuals)/d(parameter).
- weighted_residuals [numpy array] - Weighted residuals.
- weights [numpy array] - 1/(data variances normal to the best fit curve).

- WSS [float] - Weighted sum of squares residuals.
- popt [numpy array] - Optimized parameters.
- pcov [2D numpy array] - Covariance matrix of optimized parameters.
- noise_variance [float] - Estimate of the variance of the data normal to the curve.

This function is available as ``burnman.nonlinear_least_squares_fit``.

burnman.optimize.nonlinear_fitting.**confidence_prediction_bands**(*model*, *x_array*, *confidence_interval*, *f*, *flag=None*)

This function calculates the confidence and prediction bands of the function f(x) from a best-fit model with uncertainties in its parameters as calculated (for example) by the function nonlinear_least_squares_fit().

The values are calculated via the delta method, which estimates the variance of f evaluated at x as var(f(x)) = df(x)/dB var(B) df(x)/dB where df(x)/dB is the vector of partial derivatives of f(x) with respect to B.

Parameters

- **model** (*object*) – As modified (for example) by the function burnman.nonlinear_least_squares_fit(). Should contain the following functions: get_params, set_params, function, normal And attributes: delta_params, pcov, dof, noise_variance
- **x_array** (*2D numpy.array*) – Coordinates at which to evaluate the bounds.
- **confidence_interval** (*float*) – Probability level of finding the true model (confidence bound) or any new data point (probability bound). For example, the 95% confidence bounds should be calculated using a confidence interval of 0.95.
- **f** (*function*) – The function defining the variable y=f(x) for which the confidence and prediction bounds are desired.
- **flag** (*type informed by model object*) – This (optional) flag is passed to model.function to control how the modified position of x is calculated. This value is then used by f(x)

Returns

An element of bounds[i][j] gives the lower and upper confidence (i=0, i=1) and prediction (i=2, i=3) bounds for the jth data point.

Return type

2D numpy.array

burnman.optimize.nonlinear_fitting.**abs_line_project**(*M*, *n*)

burnman.optimize.nonlinear_fitting.**plot_cov_ellipse**(*cov*, *pos*, *nstd=2*, *ax=None*, ***kwargs*)

Plots an *nstd* sigma error ellipse based on the specified covariance matrix (*cov*). Additional keyword arguments are passed on to the ellipse patch artist.

Parameters

- **cov** (*numpy.array*) – The 2x2 covariance matrix to base the ellipse on.
- **pos** (*list or numpy.array*) – The location of the center of the ellipse. Expects a 2-element sequence of [x0, y0].
- **nstd** (*float*) – The radius of the ellipse in numbers of standard deviations. Defaults to 2 standard deviations.
- **ax** (*matplotlib.pyplot.axes*) – The axis that the ellipse will be plotted on. Defaults to the current axis.
- **kwargs** – Additional keyword arguments are passed on to the ellipse patch.

Returns
The covariance ellipse (already applied to the desired axes object).

Return type
matplotlib.patches.Ellipse

burnman.optimize.nonlinear_fitting.**corner_plot**(*popt*, *pcov*, *param_names=[]*, *n_std=1.0*)

Creates a corner plot of covariances

Parameters

- **popt** (*numpy.array*) – Optimized parameters.
- **pcov** (*2D numpy.array*) – Covariance matrix of the parameters.
- **param_names** (*list*) – Parameter names.
- **n_std** (*float*) – Number of standard deviations for ellipse.

Returns
`matplotlib.pyplot.figure` and list of `matplotlib.pyplot.Axes` objects.

Return type
tuple

burnman.optimize.nonlinear_fitting.**weighted_residual_plot**(*ax, model, flag=None, sd_limit=3, cmap=<matplotlib.colors.LinearSegmentedCol object>, plot_axes=[0, 1], scale_axes=[1.0, 1.0]*)

Creates a plot of the weighted residuals The user can choose the projection axes, and scaling to apply to those axes The chosen color palette (cmap) is discretised by standard deviation up to a cut off value of sd_limit.

Parameters

- **ax** – Plot.
- **type** – `matplotlib.pyplot.Axes`
- **model** (*object*) – A model as used by `burnman.nonlinear_least_squares_fit()`. Must contain the attributes model.data, model.weighted_residuals and model.flags (if flag is not None).

- **flag** (*str*) – String to determine which data to plot. Finds matches with model.flags.
- **sd_limit** (*float*) – Data with weighted residuals exceeding this limit are plotted in black.
- **cmap** (*matplotlib color palette*) – Color palette.
- **plot_axes** (*list of int*) – Data axes to use as plot axes.
- **scale_axes** (*list of float*) – Plot axes are scaled by multiplication of the data by these values.

Returns
Coloured scatter plot of the weighted residuals in data space.

Return type
matplotlib Axes object

burnman.optimize.nonlinear_fitting.**extreme_values**(*weighted_residuals*, *confidence_interval*)

This function uses extreme value theory to calculate the number of standard deviations away from the mean at which we should expect to bracket *all* of our n data points at a certain confidence level.

It then uses that value to identify which (if any) of the data points lie outside that region, and calculates the corresponding probabilities of finding a data point at least that many standard deviations away.

Parameters

- **weighted_residuals** (*array of float*) – Array of residuals weighted by the square root of their variances wr_i = r_i/sqrt(var_i).
- **confidence_interval** (*float*) – Probability at which all the weighted residuals lie within the confidence bounds.

Returns
Number of standard deviations at which we should expect to encompass all data at the user-defined confidence interval, the indices of weighted residuals exceeding the confidence_interval defined by the user, and the probabilities that the extreme data point of the distribution lies further from the mean than the observed position wr_i for each i in the "indices" output array.

Return type
tuple of (float, numpy.array, numpy.array)

burnman.optimize.nonlinear_fitting.**plot_residuals**(*ax*, *weighted_residuals*, *n_bins=None*, *flags=[]*)

burnman.optimize.nonlinear_solvers.**solve_constraint_lagrangian**(*x*, *jac_x*, *c_x*, *c_prime*)

Function which solves the problem minimize || J.dot(x_mod - x) || subject to C(x_mod) = 0 via the method of Lagrange multipliers.

Parameters

- **x** (*1D numpy array*) – Parameter values at x.

- **jac_x** (*2D numpy array.*) – The (estimated, approximate or exact) value of the Jacobian J(x).
- **c_x** (*1D numpy array*) – Values of the constraints at x.
- **c_prime** (*2D array of floats*) – The Jacobian of the constraints (A, where A.x + b = 0).

Returns

An array containing the parameter values which minimize the L2-norm of any function which has the Jacobian jac_x, and another array containing the multipliers for each of the equality constraints.

Return type

tuple

burnman.optimize.nonlinear_solvers.**damped_newton_solve**(*F*, *J*, *guess*, *tol=1e-06*, *max_iterations=100*, *lambda_bounds=<function <lambda>>*, *linear_constraints=(0.0, array([-1.]))*, *store_iterates=False*)

Solver for the multivariate nonlinear system F(x)=0 with Jacobian J(x), using the damped affine invariant modification to Newton's method (Deuflhard, 1974;1975;2004). Here we follow the algorithm as described in Nowak and Weimann (1991): [Technical Report TR-91-10, Algorithm B], modified to accept linear inequality constraints.

Linear inequality constraints are provided by the arrays constraints_A and constraints_b. The constraints are satisfied if A*x + b <= 0. If any constraints are not satisfied by the current value of lambda, lambda is reduced to satisfy all the constraints.

If a current iterate starting point (x_i) lies on one or more constraints and the Newton step violates one or more of those constraints, then the next step is calculated via the method of Lagrangian multipliers, minimizing the L2-norm of F(x_i+1) subject to the violated constraints.

Successful termination of the solver is based on three criteria:

- all(np.abs(dx (simplified newton step) < tol))
- all(np.abs(dx (full Newton step) < sqrt(10*tol))) (avoids pathology) and
- lambda = lambda_bounds(dx, x)[1] (lambda = 1 for a full Newton step).

If these criteria are not satisfied, iterations continue until one of the following occurs:

- the value of lmda is reduced to its minimum value (for v. nonlinear problems)
- successive iterations have descent vectors which violate the constraints
- the maximum number of iterations (given by max_iterations) is reached.

Information on the root (or lack of root) obtained by the solver is provided in the returned namedtuple.

Parameters

- **F** (*function of x*) – Function returning the system function F(x) as a 1D numpy array.
- **J** (*function of x*) – Function returning the Jacobian function J(x) as a 2D numpy array.
- **guess** (*1D numpy.array*) – Starting guess for the solver.
- **tol** (*float or array of floats*) – Tolerance(s) for termination.
- **max_iterations** (*int*) – Maximum number of iterations for the solver.
- **lambda_bounds** – A function of dx and x that returns a tuple of floats corresponding to the minimum and maximum allowed fractions of the full newton step (dx).
- **linear_constraints** (*tuple of a 2D numpy.array (A) and 1D numpy.array (b)*) – tuple of a 2D numpy array (A) and 1D numpy array (b) Constraints are satisfied if A.x + b < eps

Returns

A namedtuple with the following attributes:

- x: The solution vector [1D numpy array].
- F: The evaluated function F(x) [1D numpy array].
- F_norm: Euclidean norm of F(x) [float].
- J: The evaluated Jacobian J(x) [2D numpy array].
- n_it: Number of iterations [int].
- **code: Numerical description of the solver termination [int].**
  - 0: Successful convergence
  - 1: Failure due to solver hitting lower lambda bound
  - 2: Failure due to descent vector crossing constraints
  - 3: Failure due to solver reaching maximum number of iterations
- text: Description of the solver termination [str].
- success: Solver convergence state [bool].
- **iterates: [namedtuple]**
  Only present if store_iterates=True Includes the following attributes:
  - **x: list of 1D numpy arrays of floats**
    The parameters for each iteration
  - **F: list of 2D numpy arrays of floats**
    The function for each iteration
  - **lmda: list of floats**
    The value of the damping parameter for each iteration

Return type
namedtuple

6.16 Utilities

BurnMan has a number of low-level utilities to help achieve common goals. Several of these have already been described in previous sections.

6.16.1 Unit cell

burnman.utils.unitcell.**molar_volume_from_unit_cell_volume**(*unit_cell_v*, *z*)

Converts a unit cell volume from Angstroms^3 per unitcell, to m^3/mol.

Parameters

- **unit_cell_v** (*float*) – Unit cell volumes [A^3/unit cell].
- **z** (*float*) – Number of formula units per unit cell.

Returns
Volume [m^3/mol]

Return type
float

burnman.utils.unitcell.**cell_parameters_to_vectors**(*cell_parameters*)

Converts cell parameters to unit cell vectors.

Parameters
cell_parameters (*numpy.array (1D)*) – An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Returns
The three vectors defining the parallelopiped cell [m]. This function assumes that the first cell vector is colinear with the x-axis, and the second is perpendicular to the z-axis, and the third is defined in a right-handed sense.

Return type
numpy.array (2D)

burnman.utils.unitcell.**cell_vectors_to_parameters**(*M*)

Converts unit cell vectors to cell parameters.

Parameters
M (*numpy.array (2D)*) – The three vectors defining the parallelopiped cell [m]. This function assumes that the first cell vector is colinear with the x-axis, the second is perpendicular to the z-axis, and the third is defined in a right-handed sense.

Returns

An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Return type

numpy.array (1D)

6.16.2 Mathematical

burnman.utils.math.**round_to_n**(*x*, *xerr*, *n*)

burnman.utils.math.**unit_normalize**(*a*, *order=2*, *axis=-1*)

Calculates the L2 normalized array of numpy array a of a given order and along a given axis.

burnman.utils.math.**float_eq**(*a*, *b*)

Test if two floats are almost equal to each other

burnman.utils.math.**linear_interpol**(*x*, *x1*, *x2*, *y1*, *y2*)

Linearly interpolate to point x, between the points (x1,y1), (x2,y2)

burnman.utils.math.**bracket**(*fn*, *x0*, *dx*, *args=()*, *ratio=1.618*, *maxiter=100*)

Given a function and a starting guess, find two inputs for the function that bracket a root.

Parameters

- **fn** (*function*) – The function to bracket.
- **x0** (*float*) – The starting guess.
- **dx** (*float*) – Small step for starting the search.
- **args** (*tuple*) – Additional arguments to give to fn.
- **ratio** (*float*) – The step size increases by this ratio every step in the search. Defaults to the golden ratio.
- **maxiter** (*int*) – The maximum number of steps before giving up.

Returns

xa, xb, fa, fb. xa and xb are the inputs which bracket a root of fn. fa and fb are the values of the function at those points. If the bracket function takes more than maxiter steps, it raises a ValueError.

Return type

tuple of floats

burnman.utils.math.**smooth_array**(*array*, *grid_spacing*, *gaussian_rms_widths*, *truncate=4.0*, *mode='inverse_mirror'*)

Creates a smoothed array by convolving it with a gaussian filter. Grid resolutions and gaussian RMS

widths are required for each of the axes of the numpy array. The smoothing is truncated at a user-defined number of standard deviations. The edges of the array can be padded in a number of different ways given by the 'mode' parameter.

Parameters

- **array** (*numpy.ndarray*) – The array to smooth.
- **grid_spacing** (*numpy.array of floats*) – The spacing of points along each axis.
- **gaussian_rms_widths** (*numpy.array of floats*) – The Gaussian RMS widths/standard deviations for the Gaussian convolution.
- **truncate** (*float*) – The number of standard deviations at which to truncate the smoothing.
- **mode** (*str*) – {'reflect', 'constant', 'nearest', 'mirror', 'wrap', 'inverse_mirror'} The mode parameter determines how the array borders are handled either by scipy.ndimage.filters.gaussian_filter. Default is 'inverse_mirror', which uses `burnman.tools.math._pad_ndarray_inverse_mirror()`.

Returns
The smoothed array

Return type
numpy.ndarray

`burnman.utils.math.`**interp_smoothed_array_and_derivatives**(*array*, *x_values*, *y_values*, *x_stdev=0.0*, *y_stdev=0.0*, *truncate=4.0*, *mode='inverse_mirror'*, *indexing='xy'*)

Creates a smoothed array on a regular 2D grid. Smoothing is achieved using `burnman.tools.math.smooth_array()`. Outputs scipy.interpolate.RegularGridInterpolator() interpolators which can be used to query the array, or its derivatives in the x- and y- directions.

Parameters

- **array** (*numpy.array (2D)*) – The array to smooth. Each element array[i][j] corresponds to the position x_values[i], y_values[j]
- **x_values** (*numpy.array (1D)*) – The gridded x values over which to create the smoothed grid
- **y_values** (*numpy.array (1D)*) – The gridded y_values over which to create the smoothed grid
- **x_stdev** (*float*) – The standard deviation for the Gaussian filter along the x axis
- **y_stdev** (*float*) – The standard deviation for the Gaussian filter along the y axis

- **truncate** (*float*) – The number of standard deviations at which to truncate the smoothing.
- **mode** (*str*) – {'reflect', 'constant', 'nearest', 'mirror', 'wrap', 'inverse_mirror'} The mode parameter determines how the array borders are handled either by scipy.ndimage.filters.gaussian_filter. Default is 'inverse_mirror', which uses `burnman.tools.math._pad_ndarray_inverse_mirror()`.
- **indexing** (*str*) – {'xy', 'ij'}, optional Cartesian ('xy', default) or matrix ('ij') indexing of output. See numpy.meshgrid for more details.

Returns
Three RegularGridInterpolator functors interpolation functions for the smoothed property and the first derivatives with respect to x and y.

Return type
tuple

`burnman.utils.math.compare_l2(depth, calc, obs)`

Computes the L2 norm for N profiles at a time (assumed to be linear between points).

Parameters

- **depths** (*array of float*) – depths. $[m]$
- **calc** (*list of arrays of float*) – N arrays calculated values, e.g. [mat_vs,mat_vphi]
- **obs** (*list of arrays of float*) – N arrays of values (observed or calculated) to compare to , e.g. [seis_vs, seis_vphi]

Returns
array of L2 norms of length N

Return type
array of floats

`burnman.utils.math.compare_chifactor(calc, obs)`

Computes the chi factor for N profiles at a time. Assumes a 1% a priori uncertainty on the seismic model.

Parameters

- **calc** (*list of arrays of float*) – N arrays calculated values, e.g. [mat_vs,mat_vphi]
- **obs** (*list of arrays of float*) – N arrays of values (observed or calculated) to compare to , e.g. [seis_vs, seis_vphi]

Returns
error array of length N

Return type
array of floats

burnman.utils.math.**l2**(*x*, *funca*, *funcb*)

Computes the L2 norm for one profile(assumed to be linear between points).

Parameters

- **x** (*array of float*) – depths $[m]$.
- **funca** (*list of arrays of float*) – array calculated values
- **funcb** (*list of arrays of float*) – array of values (observed or calculated) to compare to

Returns
L2 norm

Return type
array of floats

burnman.utils.math.**nrmse**(*x*, *funca*, *funcb*)

Normalized root mean square error for one profile :type x: array of float :param x: depths in m. :type funca: list of arrays of float :param funca: array calculated values :type funcb: list of arrays of float :param funcb: array of values (observed or calculated) to compare to

Returns
RMS error

Return type
array of floats

burnman.utils.math.**chi_factor**(*calc*, *obs*)

χ factor for one profile assuming 1% uncertainty on the reference model (obs) :type calc: list of arrays of float :param calc: array calculated values :type obs: list of arrays of float :param obs: array of reference values to compare to

Returns
χ factor

Return type
array of floats

burnman.utils.math.**independent_row_indices**(*array*)

Returns the indices corresponding to an independent set of rows for a given array. The independent rows are determined from the pivots used during row reduction/Gaussian elimination.

Parameters
array (*2D numpy.array of floats*) – The input array.

Returns
The indices corresponding to a set of independent rows of the input array.

Return type
1D numpy array of integers

burnman.utils.math.**array_to_rational_matrix**(*array*)

Converts a numpy array into a sympy matrix filled with rationals

burnman.utils.math.**generate_complete_basis**(*incomplete_basis*, *array*)

Given a 2D array with independent rows and a second 2D array that spans a larger space, creates a complete basis for the combined array using all the rows of the first array, followed by any required rows of the second array. So, for example, if the first array is: [[1, 0, 0], [1, 1, 0]] and the second array is: [[1, 0, 0], [0, 1, 0], [0, 0, 1]], the complete basis will be: [[1, 0, 0], [1, 1, 0], [0, 0, 1]].

Parameters

- **incomplete_basis** (*2D numpy.array*) – An array containing the basis to be completed.
- **array** (*2D numpy.array*) – An array spanning the full space for which a basis is required.

Returns

An array containing the basis vectors spanning both of the input arrays.

Return type

2D numpy array

6.16.3 Miscellaneous

class burnman.utils.misc.**OrderedCounter**(*iterable=None*, */*, ***kwds*)

Bases: Counter, OrderedDict

Counter that remembers the order elements are first encountered

clear() → None. Remove all items from od.

copy()

Return a shallow copy.

elements()

Iterator over elements repeating each as many times as its count.

```
>>> c = Counter('ABCABC')
>>> sorted(c.elements())
['A', 'A', 'B', 'B', 'C', 'C']
```

Knuth's example for prime factors of 1836: 2**2 * 3**3 * 17**1 >>> import math >>> prime_factors = Counter({2: 2, 3: 3, 17: 1}) >>> math.prod(prime_factors.elements()) 1836

Note, if an element's count has been set to zero or is a negative number, elements() will ignore it.

classmethod fromkeys(*iterable*, *v=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

most_common(*n=None*)

List the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

move_to_end(*/*, *key*, *last=True*)

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*/*, *key*, *default=<unrepresentable>*)

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem(*/*, *last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*/*, *key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

subtract(*iterable=None*, */*, ***kwds*)

Like dict.update() but subtracts counts instead of replacing them. Counts can be reduced below zero. Both the inputs and outputs are allowed to contain zero and negative counts.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.subtract('witch')             # subtract elements from another␣
↪iterable
>>> c.subtract(Counter('watch'))    # subtract elements from another␣
↪counter
>>> c['h']                          # 2 in which, minus 1 in witch,␣
↪minus 1 in watch
0
>>> c['w']                          # 1 in which, minus 1 in witch,␣
↪minus 1 in watch
-1
```

total()

Sum of the counts

update(*iterable=None*, */*, ***kwds*)

Like dict.update() but add counts instead of replacing them.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.update('witch')           # add elements from another iterable
>>> d = Counter('watch')
>>> c.update(d)                 # add elements from another counter
>>> c['h']                      # four 'h' in which, witch, and watch
4
```

values() → an object providing a view on D's values

burnman.utils.misc.**copy_documentation**(*copy_from*)

Decorator @copy_documentation(another_function) will copy the documentation found in a different function (for example from a base class). The docstring applied to some function a() will be

```
(copied from BaseClass.some_function):
<documentation from BaseClass.some_function>
<optionally the documentation found in a()>
```

burnman.utils.misc.**merge_two_dicts**(*x*, *y*)

Given two dicts, merge them into a new dict as a shallow copy.

burnman.utils.misc.**flatten**(*arr*)

burnman.utils.misc.**pretty_print_values**(*popt*, *pcov*, *params*)

Takes a numpy array of parameters, the corresponding covariance matrix and a set of parameter names and prints the parameters and principal 1-s.d.uncertainties (np.sqrt(pcov[i][i])) in a nice text based format.

burnman.utils.misc.**pretty_print_table**(*table*, *use_tabs=False*)

Takes a 2d table and prints it in a nice text based format. If use_tabs=True then only is used as a separator. This is useful for importing the data into other apps (Excel, ...). The default is to pad the columns with spaces to make them look neat. The first column is left aligned, while the remainder is right aligned.

burnman.utils.misc.**sort_table**(*table*, *col=0*)

Sort the table according to the column number

burnman.utils.misc.**read_table**(*filename*)

burnman.utils.misc.**array_from_file**(*filename*)

Generic function to read a file containing floats and commented lines into a 2D numpy array.

Commented lines are prefixed by the characters # or %.

burnman.utils.misc.**cut_table**(*table*, *min_value*, *max_value*)

burnman.utils.misc.**lookup_and_interpolate**(*table_x*, *table_y*, *x_value*)

burnman.utils.misc.**attribute_function**(*m*, *attributes*, *powers=[]*)

Function which returns a function which can be used to evaluate material properties at a point. This function allows the user to define the property returned as a string. The function can itself be passed to another function (such as nonlinear_fitting.confidence_prediction_bands()).

Properties can either be simple attributes (e.g. K_T) or a product of attributes, each raised to some power.

Parameters

- **m** (*burnman.Material*) – The material instance evaluated by the output function.
- **attributes** (*list of str*) – The list of material attributes / properties to be evaluated in the product.
- **powers** (*list of floats*) – The powers to which each attribute should be raised during evaluation.

Returns

Function which returns the value of product(a_i**p_i) as a function of condition (x = [P, T, V]).

Return type

function

6.17 Tools

Burnman has a number of high-level tools to help achieve common goals. Several of these have already been described in previous sections.

6.17.1 Plotting

burnman.tools.plot.**pretty_plot**()

Makes pretty plots. Overwrites the matplotlib default settings to allow for better fonts. Slows down plotting

burnman.tools.plot.**plot_projected_elastic_properties**(*mineral*, *plot_types*, *axes*, *n_zenith=31*, *n_azimuth=91*, *n_divs=100*)

Parameters

- **mineral** (*burnman.Mineral*) – Mineral object on which calculations should be done
- **plot_types** (*list of str*) – Plot types must be one of the following * ‘vp’ - V$_{P}$ (km/s) * ‘vs1’ - ‘V$_{S1}$ (km/s) * ‘vs2’ - V$_{S2}$ (km/s) * ‘vp/vs1’ - V$_{P}$/V$_{S1}$ * ‘vp/vs2’ - V$_{P}$/V$_{S2}$ * ‘s

anisotropy' - S-wave anisotropy (%), 200(vs1s - vs2s)/(vs1s + vs2s)) * 'linear compressibility' - Linear compressibility (GPa^{-1}) * 'youngs modulus' - Youngs Modulus (GPa)

- **axes** (*matplotlib.pyplot.axes objects*) – axes objects to be modified. Must be initialised with projection='polar'.
- **n_zenith** (*int*) – Number of zeniths (plot resolution).
- **n_azimuth** (*int*) – Number of azimuths (plot resolution).
- **n_divs** (*int*) – Number of divisions for the color scale.

6.17.2 Output for seismology

burnman.tools.output_seismo.**write_tvel_file**(*planet_or_layer*, *modelname='burnmanmodel'*, *background_model=None*)

Function to write input file for obspy travel time calculations. Note: Because density isn't defined for most 1D seismic models, densities are output as zeroes. The tvel format has a column for density, but this column is not used by obspy for travel time calculations.

Parameters

- **planet_or_layer** (*burnman.Planet* or *burnman.Layer*.) – Planet or layer to write out to tvel file
- **filename** (*str*) – Filename to read to.
- **background_model** (burnman.seismic.Seismic1DModel) – 1D seismic model to fill in parts of planet (likely to be an earth model) that aren't defined by layer (only need when using *burnman.Layer*)

burnman.tools.output_seismo.**write_axisem_input**(*layers*, *modelname='burnmanmodel_foraxisem'*, *axisem_ref='axisem_prem_ani_noocean.txt'*, *plotting=False*)

Writing velocities and densities to AXISEM (www.axisem.info) input file. The input can be a single layer, or a list of layers taken from a planet (planet.layers). Currently this function will implement explicit discontinuities between layers in the seismic model. Currently this function is only set for Earth.

Parameters

- **layers** (list of one or more *burnman.Layer*) – List of layers to put in AXISEM file.
- **modelname** (*str*) – Name of model, appears in name of output file.
- **axisem_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

burnman.tools.output_seismo.**write_mineos_input**(*layers*, *modelname='burnmanmodel_formineos'*, *mineos_ref='mineos_prem_noocean.txt'*, *plotting=False*)

Writing velocities and densities to Mineos (https://geodynamics.org/cig/software/mineos/) input file Note: currently, this function only honors the discontinuities already in the synthetic input file, so it is best to only replace certain layers with burnman values

Parameters

- **layers** (list of one or more *burnman.Layer*) – List of layers to put in Mineos file.
- **modelname** (*str*) – Name of model, appears in name of output file.
- **mineos_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

6.17.3 Equations of state

burnman.tools.eos.**check_eos_consistency**(*m*, *P=1000000000.0*, *T=300.0*, *tol=0.0001*, *verbose=False*, *including_shear_properties=True*)

Checks that numerical derivatives of the Gibbs energy of a mineral under given conditions are equal to those provided analytically by the equation of state.

Parameters

- **m** (*burnman.Mineral*) – The mineral for which the equation of state is to be checked for consistency.
- **P** (*float*) – The pressure at which to check consistency.
- **T** (*float*) – The temperature at which to check consistency.
- **tol** (*float*) – The fractional tolerance for each of the checks.
- **verbose** (*bool*) – Decide whether to print information about each check.
- **including_shear_properties** (*bool*) – Decide whether to check shear information, which is pointless for liquids and equations of state without shear modulus parameterizations.

Returns

Boolean stating whether all checks have passed.

Return type

bool

burnman.tools.eos.**check_anisotropic_eos_consistency**(*m*, *P=1000000000.0*, *T=2000.0*, *tol=0.0001*, *verbose=False*)

Checks that numerical derivatives of the Gibbs energy of an anisotropic mineral under given conditions are equal to those provided analytically by the equation of state.

Parameters

- **m** (*burnman.AnisotropicMineral*) – The anisotropic mineral for which the equation of state is to be checked for consistency.
- **P** (*float*) – The pressure at which to check consistency.
- **T** (*float*) – The temperature at which to check consistency.
- **tol** (*float*) – The fractional tolerance for each of the checks.
- **verbose** (*bool*) – Decide whether to print information about each check.

Returns

Boolean stating whether all checks have passed.

Return type

bool

CHAPTER

SEVEN

CHANGES

The following is a list of recent improvements to BurnMan.

- BurnMan 0.9.0 is released.

  The BurnMan Team, 2021/08/02

- The BurnMan homepage is updated and moved to https://geodynamics.github.io/burnman/

  Bob Myhill and Timo Heister, 2021/08/04

- `burnman.composite.Composite` now has new properties which include the *endmember_formulae*, a list of *elements* which make up those formulae, the *stoichiometric_matrix* (number of atoms of element j in formula i), and an independent *reaction_basis*. These properties are calculated once when they are first needed, and then cached.

  Bob Myhill, 2021/08/05

- BurnMan now has a new (experimental) function, `burnman.equilibrate`, which allows users to calculate the equilibrium pressure, temperature, phase proportions and compositions given two constraints. Several examples are provided in the file examples/example_equilibrate.py.

  Bob Myhill, 2021/09/27

- BurnMan now has a new anisotropic equation of state class, *burnman.AnisotropicMineral*, which can be used to model materials of arbitrary symmetry under hydrostatic conditions. Users can set the state (pressure and temperature) of AnisotropicMineral objects and then retrieve their anisotropic properties. Details of the formulation can be found in [Myhill22]. Examples are provided in the file examples/example_anisotropic_mineral.py.

  Bob Myhill, 2021/10/03

- The experimental function `burnman.equilibrate` now allows users to calculate equilibrium assemblage properties while allowing the bulk composition to vary. An example is provided in the file examples/example_olivine_binary.py.

  Bob Myhill, 2021/10/08

- The ideal solution model has been updated to allow site multiplicities to vary linearly with endmember proportions. It therefore implements a Temkin-type model [Tem45].

- As a result of the changes to the ideal model, the site_multiplicities attribute of a solid solution is now a 2D array.

Bob Myhill, 2021/10/22

- A new BoundaryLayerPerturbation class has been implemented that allows the user to create a thermal perturbation to a planetary layer according to the model proposed by [RM81]. This perturbation is exponential, taking the form: T = a*exp((r - r1)/(r0 - r1)*c) + b*exp((r - r0)/(r1 - r0)*c). The user defines a, b and c using three related parameters: rayleigh_number (Ra = c^4), temperature_change (the total difference in temperature perturbation across the layer, (a + b)*exp(c)), and boundary_layer_ratio (a/b). Once initialised, a BoundaryLayerPerturbation object can be modified using the function set_model_thermal_gradients, which sets the thermal gradient (dT/dr) at the bottom and top of the layer).

  Bob Myhill, 2021/10/22

- A new equation of state, BroshCalphad has been implemented, as described in [BMS07]. This new equation of state is used to define the properties of the BCC, FCC, HCP and liquid polymorphs of iron, as parameterised by [SE15]. These endmembers can be initialised from the BurnMan mineral database (*bcc_iron*, *fcc_iron*, *hcp_iron* and *liquid_iron*).

  Bob Myhill, 2021/10/22

- The *burnman.Composite* class now has a *chemical_potential* method. This replaces the more limited function `burnman.tools.chemistry.chemical_potentials()`. The related functions *burnman.tools.chemistry.fugacity()* and *burnman.tools.chemistry.relative_fugacity()* have been updated to use this method, and to take Composites as arguments, rather than lists of Minerals.

  Bob Myhill, 2021/10/30

- Running the examples now requires BurnMan to be installed. This change is made to ensure that tester behaviour always matches installed behaviour, rather than local behaviour (the cause of the problem with the 1.0.0 release). If the user wishes to develop the BurnMan code, they may install the module from the top-level directory using pip with the *-e* (editable) flag: *python -m pip install -e ..*

  Bob Myhill, 2021/11/02

- BurnMan now has a Calibrant class. This class is a stripped-down version of a Material that is initialised with a params dictionary and a function to compute the pressure or volume at given conditions.

  Objects derived from this class have the ability to output pressure/volume as a function of volume/pressure and temperature. The user can pass a V-T or P-T covariance matrix as an additional optional argument, in which case BurnMan will propagate the errors and output a full P-V-T or V-P-T covariance matrix.

  A helper function is provided to convert from pressure calculated using one calibration into pressure calculated using another calibration of the same material.

  This class is expected to be used by experimental petrologists and those interpreting high pressure experimental data.

  Bob Myhill, 2022/01/28

- New: The EoS fitting functions now have additional (optional) arguments. delta_params allows the user to set initial values for the change in parameters (i.e. initial step size). bounds allows the user to set hard bounds on the values of each of the parameters.

  Bob Myhill, 2022/02/05

- New: BurnMan now has a new fitting function, *burnman.optimize.eos_fitting.fit_XPTp_data()*. It has the same arguments as *burnman.optimize.eos_fitting.fit_PTp_data()*, but instead of fitting data to a mineral of fixed composition, it is able to simultaneously fit parameters of solution model parameters, including equation of state parameters for each endmember.

  Bob Myhill, 2022/02/05

- Changed: The default name of the solution class has been changed from `burnman.classes.SolidSolution` to `burnman.classes.Solution`. An alias has been provided for backwards compatibility.

  Bob Myhill, 2022/02/07

- Changed: BurnMan now has another submodule called utils. Many of the functions that used to be in the tools submodule have now been moved into the utils submodule. Functions residing in utils do not depend on other burnman submodules (in rare cases they depend on objects defined in constants.py). Functions residing in tools are not called by other submodules in BurnMan. Thus, utils is a relatively low level submodule, while tools is a high level submodule. This change has been made to avoid future issues with circular imports.

  Bob Myhill, 2022/02/16

- New: The BurnMan Material classes now have a method called set_state_with_volume that sets state using volume and temperature rather than pressure and temperature.

  Bob Myhill, 2022/02/16

- New: BurnMan now has an ElasticSolution class that implements the elastic solution proposed in [Myhill18]. This class defines excess thermodynamic properties of solutions relative to the properties of the solution endmembers at fixed volume and temperature, rather than at fixed pressure and temperature. Thus, the models are most easily expressed through parameterisations of the Helmholtz energy, rather than the Gibbs energy.

  Bob Myhill, 2022/06/24

- New: BurnMan now has a function method for defining solution/elastic solution models. To use this method, the user should provide a function for the excess nonconfigurational Gibbs/Helmholtz energy during initialisation of the model. The ideal entropy is calculated internally as for all the other solution models.

  Bob Myhill, 2022/06/25

- BurnMan now has two new helper functions: *burnman.tools.chemistry.reactions_from_stoichiometric_matrix()* *burnman.tools.chemistry.reactions_from_formulae()*. These functions generate a complete list of reactions (forward and reverse) from either the stoichiometric matrix (a 2D numpy array containing the amount of component j in phase i), or from a list of formulae (as strings or dictionaries of elemental amounts).

  Bob Myhill, 2022/10/03

- BurnMan Solution and ElasticSolution objects are now instantiated with a SolutionModel object as a solution_model parameter. The use of the solution_type parameter has been removed completely, along with all of the optional parameters that were originally passed as parameters to SolutionModel.

Bob Myhill, 2022/10/22

- BurnMan now has a generalised PolynomialSolution class. The non-ideal excesses in this model are polynomial functions of composition. The coefficients of these polynomials are either constant internal energy, entropy and volume (i.e. Gxs = (Eijk... - T Sijk... + P Vijk...) xi xj xk...) or a linear sum of Mineral objects (i.e. Gxs = amijk mineralm.gibbs ... xi xj xk ...). Coefficients for both of these forms are passed as a list of lists to instantiate a PolynomialSolution object, which can then be passed as usual to instantiate a Solution object.

  Optionally, a transformation matrix can be passed that allows users to define the coefficients above as functions of a modified set of endmember proportions: p'i = Aij pj. This may be useful when dealing with ordering, as expressing excess terms as a function of order parameter is often much more illuminating than expressing them in terms of endmember proportions.

  This class can deal with arbitrarily high powers in endmember proportions. However, because the class internally converts the list of lists to numpy arrays, high powers of solutions with a large number of endmembers will create very large arrays (with order n_endmembers^(highest power) elements). This may significantly slow down calculations.

  For most purposes, using dense numpy arrays is much faster than using sparse arrays in COO format (a downside of using python). Should users need particularly complex solutions with high power terms, a modified solution model will be required.

  Bob Myhill, 2022/10/22

- Four new property modifier formulations are provided, which can be specified with the names "debye", "debye_delta", "einstein" and "einstein_delta". These are based on the Debye and Einstein models of thermal energy. The heat capacity ("debye", "einstein") or entropy ("debye_delta", "einstein_delta") are based on the heat capacity of the respective thermal model, and so reach a maximum at high temperature. The excess energy, entropy and heat capacity are all zero at 0 K. The excess volume is always zero.

  Bob Myhill, 2022/10/23

BIBLIOGRAPHY

[And82] O. L. Anderson. The Earth's Core and the Phase Diagram of Iron. *Philos. T. Roy. Soc. A*, 306(1492):21–35, 1982. URL: http://rsta.royalsocietypublishing.org/content/306/1492/21.abstract.

[ASA+11] D Antonangeli, J Siebert, CM Aracne, D Farber, A Bosak, M Hoesch, M Krisch, F Ryerson, G Fiquet, and J Badro. Spin crossover in ferropericlase at high pressure: a seismologically transparent transition? *Science*, 331(6031):64–67, 2011. URL: http://www.sciencemag.org/content/331/6013/64.short.

[BM67] T. H. K. Barron and R. W. Munn. Analysis of the thermal expansion of anisotropic solids: application to zinc. *The Philosophical Magazine: A Journal of Theoretical Experimental and Applied Physics*, 15(133):85–103, 1967. URL: https://doi.org/10.1080/14786436708230352, arXiv:https://doi.org/10.1080/14786436708230352, doi:10.1080/14786436708230352.

[BMS07] Eli Brosh, Guy Makov, and Roni Z. Shneck. Application of CALPHAD to high pressures. *Calphad*, 31(2):173–185, 2007. URL: https://www.sciencedirect.com/science/article/pii/S0364591607000168, doi:https://doi.org/10.1016/j.calphad.2006.12.008.

[BS81] JM Brown and TJ Shankland. Thermodynamic parameters in the Earth as determined from seismic profiles. *Geophys. J. Int.*, 66(3):579–596, 1981. URL: http://gji.oxfordjournals.org/content/66/3/579.short.

[CGDG05] F Cammarano, S Goes, A Deuss, and D Giardini. Is a pyrolitic adiabatic mantle compatible with seismic data? *Earth Planet. Sci. Lett.*, 232(3):227–243, 2005. URL: http://www.sciencedirect.com/science/article/pii/S0012821X05000804.

[Cam13] F. Cammarano. A short note on the pressure-depth conversion for geophysical interpretation. *Geophysical Research Letters*, 40(18):4834–4838, 2013. URL: https://doi.org/10.1002/grl.50887, doi:10.1002/grl.50887.

[CHS87] CP Chin, S Hertzman, and B Sundman. An evaluation of the composition dependence of the magnetic order-disorder transition in cr-fe-co-ni alloys. *Materials Research Center, The Royal Institute of Technology (Stockholm, Sweden), Report TRITA-MAC*, 1987.

[CGR+09] L Cobden, S Goes, M Ravenna, E Styles, F Cammarano, K Gallagher, and JA Connolly. Thermochemical interpretation of 1-D seismic data for the lower mantle: The significance of nonadiabatic thermal gradients and compositional heterogeneity. *J. Geophys. Res.*, 114:B11309, 2009. URL: http://www.agu.org/journals/jb/jb0911/2008JB006262/2008jb006262-t01.txt.

[Con05] JAD Connolly. Computation of phase equilibria by linear programming: a tool for geodynamic modeling and its application to subduction zone decarbonation. *Earth Planet. Sci. Lett.*, 236(1):524–541, 2005. URL: http://www.sciencedirect.com/science/article/pii/S0012821X05002839.

[CHRU14] Sanne Cottaar, Timo Heister, Ian Rose, and Cayman Unterborn. Burnman: a lower mantle mineral physics toolkit. *Geochemistry, Geophysics, Geosystems*, 15(4):1164–1179, 2014. URL: https://doi.org/10.1002/2013GC005122, doi:10.1002/2013GC005122.

[DGD+12] DR Davies, S Goes, JH Davies, BSA Shuberth, H-P Bunge, and J Ritsema. Reconciling dynamic and seismic models of Earth's lower mantle: The dominant role of thermal heterogeneity. *Earth Planet. Sci. Lett.*, 353:253–269, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X1200444X.

[DCT12] F Deschamps, L Cobden, and PJ Tackley. The primitive nature of large low shear-wave velocity provinces. *Earth Planet. Sci. Lett.*, 349-350:198–208, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X12003718.

[DT03] Frédéric Deschamps and Jeannot Trampert. Mantle tomography and its relation to temperature and composition. *Phys. Earth Planet. Int.*, 140(4):277–291, December 2003. URL: http://www.sciencedirect.com/science/article/pii/S0031920103001894, doi:10.1016/j.pepi.2003.09.004.

[DPWH07] J. F. A. Diener, R. Powell, R. W. White, and T. J. B. Holland. A new thermodynamic model for clino- and orthoamphiboles in the system Na$_2$O–CaO–FeO–MgO–Al$_2$O$_3$–SiO$_2$–H$_2$O–O. *Journal of Metamorphic Geology*, 25(6):631–656, 2007. URL: https://doi.org/10.1111/j.1525-1314.2007.00720.x, doi:10.1111/j.1525-1314.2007.00720.x.

[DA81] A M Dziewonski and D L Anderson. Preliminary reference Earth model. *Phys. Earth Planet. Int.*, 25(4):297–356, 1981.

[HW12] Y He and L Wen. Geographic boundary of the "Pacific Anomaly" and its geometry and transitional structure in the north. *J. Geophys. Res.-Sol. Ea.*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2012JB009436/full, doi:DOI: 10.1029/2012JB009436.

[HW89] George Helffrich and Bernard J Wood. Subregular model for multicomponent solutions. *American Mineralogist*, 74(9-10):1016–1022, 1989.

[HernandezAlfeB13] ER Hernández, D Alfè, and J Brodholt. The incorporation of water into lower-mantle perovskites: A first-principles study. *Earth Planet. Sci. Lett.*, 364:37–43, 2013. URL: http://www.sciencedirect.com/science/article/pii/S0012821X13000137.

[HHPH13a] T. J. B. Holland, N. F. C. Hudson, R. Powell, and B. Harte. New Thermodynamic Models and Calculated Phase Equilibria in NCFMAS for Basic and Ultrabasic Compositions through the Transition Zone into the Uppermost Lower Mantle. *Journal of Petrology*, 54(9):1901–1920, July 2013. URL: http://petrology.oxfordjournals.org/content/54/9/1901.short, doi:10.1093/petrology/egt035.

[HP90] T. J. B. Holland and R. Powell. An enlarged and updated internally consistent thermodynamic dataset with uncertainties and correlations: the system K2O-Na$_2$O-CaO-MgO-MnO-FeO-Fe$_2$O$_3$-Al$_2$O$_3$-TiO$_2$-SiO$_2$-C-H$_2$-O$_2$. *Journal of Metamorphic Geology*, 8(1):89–124, 1990. URL: https://doi.org/10.1111/j.1525-1314.1990.tb00458.x, doi:10.1111/j.1525-1314.1990.tb00458.x.

[HP06] T. J. B. Holland and R. Powell. Mineral activity–composition relations and petrological calculations involving cation equipartition in multisite minerals: a logical inconsistency. *Journal of Metamorphic Geology*, 24(9):851–861, 2006. URL: https://doi.org/10.1111/j.1525-1314.2006.00672.x, doi:10.1111/j.1525-1314.2006.00672.x.

[HP91] Tim Holland and Roger Powell. A Compensated-Redlich-Kwong (CORK) equation for volumes and fugacities of CO2 and H2O in the range 1 bar to 50 kbar and 100–1600°C. *Contributions to Mineralogy and Petrology*, 109(2):265–273, 1991. URL: https://doi.org/10.1007/BF00306484, doi:10.1007/BF00306484.

[HP96] Tim Holland and Roger Powell. Thermodynamics of order-disorder in minerals; ii, symmetric formalism applied to solid solutions. *American Mineralogist*, 81(11-12):1425–1437, 1996. URL: http://ammin.geoscienceworld.org/content/81/11-12/1425, arXiv:http://ammin.geoscienceworld.org/content/81/11-12/1425, doi:10.2138/am-1996-11-1215.

[HHPH13b] Tim J.B. Holland, Neil F.C. Hudson, Roger Powell, and Ben Harte. New thermodynamic models and calculated phase equilibria in NCFMAS for basic and ultrabasic compositions through the transition zone into the uppermost lower mantle. *Journal of Petrology*, 54(9):1901–1920, 2013. URL: http://petrology.oxfordjournals.org/content/54/9/1901.abstract, arXiv:http://petrology.oxfordjournals.org/content/54/9/1901.full.pdf+html, doi:10.1093/petrology/egt035.

[HMSL08] C Houser, G Masters, P Shearer, and G Laske. Shear and compressional velocity models of the mantle from cluster analysis of long-period waveforms. *Geophys. J. Int.*, 174(1):195–212, 2008.

[IWSY10] T Inoue, T Wada, R Sasaki, and H Yurimoto. Water partitioning in the Earth's mantle. *Phys. Earth Planet. Int.*, 183(1):245–251, 2010. URL: http://www.sciencedirect.com/science/article/pii/S0031920110001573.

[IS92] Joel Ita and Lars Stixrude. Petrology, elasticity, and composition of the mantle transition zone. *Journal of Geophysical Research*, 97(B5):6849, 1992. URL: http://doi.wiley.com/10.1029/92JB00068, doi:10.1029/92JB00068.

[Jac98] Ian Jackson. Elasticity, composition and temperature of the Earth's lower mantle: a reappraisal. *Geophys. J. Int.*, 134(1):291–311, July 1998. URL: http://gji.oxfordjournals.org/content/134/1/291.abstract, doi:10.1046/j.1365-246x.1998.00560.x.

[JCK+10] Matthew G Jackson, Richard W Carlson, Mark D Kurz, Pamela D Kempton, Don Francis, and Jerzy Blusztajn. Evidence for the survival of the oldest terrestrial mantle reservoir. *Nature*, 466(7308):853–856, 2010.

[KS90] SI Karato and HA Spetzler. Defect microdynamics in minerals and solid-state mechanisms of seismic wave attenuation and velocity dispersion in the mantle. *Rev. Geophys.*, 28(4):399–421, 1990. URL: http://onlinelibrary.wiley.com/doi/10.1029/RG028i004p00399/full.

[Kea54] A Keane. An Investigation of Finite Strain in an Isotropic Material Subjected to Hydrostatic Pressure and its Seismological Applications. *Australian Journal of Physics*, 7(2):322, 1954. URL: http://www.publish.csiro.au/?paper=PH540322, doi:10.1071/PH540322.

[KEB95] BLN Kennett, E R Engdahl, and R Buland. Constraints on seismic velocities in the Earth from traveltimes. *Geophys. J. Int.*, 122(1):108–124, 1995. URL: http://gji.oxfordjournals.org/content/122/1/108.short.

[KE91] BLN Kennett and ER Engdahl. Traveltimes for global earthquake location and phase identification. *Geophysical Journal International*, 105(2):429–465, 1991.

[KHM+12] Y Kudo, K Hirose, M Murakami, Y Asahara, H Ozawa, Y Ohishi, and N Hirao. Sound velocity measurements of CaSiO_3 perovskite to 133 GPa and implications for lowermost mantle seismic anomalies. *Earth Planet. Sci. Lett.*, 349:1–7, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X1200324X.

[KED08] B Kustowski, G Ekstrom, and AM Dziewonski. Anisotropic shear-wave velocity structure of the Earth's mantle: a global model. *J. Geophys. Res.*, 113(B6):B06306, 2008. URL: http://www.agu.org/pubs/crossref/2008/2007JB005169.shtml.

[LCDR12] V Lekic, S Cottaar, A M Dziewonski, and B Romanowicz. Cluster analysis of global lower mantle tomography: A new class of structure and implications for chemical heterogeneity. *Earth Planet. Sci. Lett.*, 357-358:68–77, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X12005109.

[LvdH08] C Li and RD van der Hilst. A new global model for P wave speed variations in Earth's mantle. *Geochem. Geophys. Geosyst.*, 2008. URL: http://onlinelibrary.wiley.com/doi/10.1029/2007GC001806/full.

[LSMM13] JF Lin, S Speziale, Z Mao, and H Marquardt. Effects of the electronic spin transitions of iron in lower mantle minerals: Implications for deep mantle geophysics and geochemistry. *Rev. Geophys.*, 2013. URL: http://onlinelibrary.wiley.com/doi/10.1002/rog.20010/full.

[LVJ+07] Jung-Fu Lin, György Vankó, Steven D. Jacobsen, Valentin Iota, Viktor V. Struzhkin, Vitali B. Prakapenka, Alexei Kuznetsov, and Choong-Shik Yoo. Spin transition zone in Earth's lower mantle. *Science*, 317(5845):1740–1743, 2007. URL: http://www.sciencemag.org/content/317/5845/1740.abstract, arXiv:http://www.sciencemag.org/content/317/5845/1740.full.pdf.

[MegninR00] C Mégnin and B Romanowicz. The three-dimensional shear velocity structure of the mantle from the inversion of body, surface and higher-mode waveforms. *Geophys. J. Int.*, 143(3):709–728, 2000.

[MHS11] David Mainprice, Ralf Hielscher, and Helmut Schaeben. Calculating anisotropic physical properties from texture data using the MTEX open-source package. *Geological Society, London, Special Publications*, 360(1):175–192, 2011.

[MLS+11] Z Mao, JF Lin, HP Scott, HC Watson, VB Prakapenka, Y Xiao, P Chow, and C McCammon. Iron-rich perovskite in the Earth's lower mantle. *Earth Planet. Sci. Lett.*, 309(3):179–184, 2011. URL: http://www.sciencedirect.com/science/article/pii/S0012821X11004018.

[MWF11] G. Masters, J.H. Woodhouse, and G. Freeman. Mineos v1.0.2 [software]. *Computational Infrastructure for Geodynamics*, :99, 2011. URL: https://geodynamics.org/cig/software/mineos/.

[MBR+07] J Matas, J Bass, Y Ricard, E Mattern, and MST Bukowinski. On the bulk composition of the lower mantle: predictions and limitations from generalized inversion of radial seismic profiles. *Geophys. J. Int.*, 170(2):764–780, 2007. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2007.03454.x/full.

[MB07] J Matas and MST Bukowinski. On the anelastic contribution to the temperature dezpendence of lower mantle seismic velocities. *Earth Planet. Sci. Lett.*, 259(1):51–65, 2007. URL: http://www.sciencedirect.com/science/article/pii/S0012821X07002555.

[MMRB05] E. Mattern, J. Matas, Y. Ricard, and J. Bass. Lower mantle composition and temperature from mineral physics and thermodynamic modelling. *Geophys. J. Int.*, 160(3):973–990, March 2005. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2004.02549.x, doi:10.1111/j.1365-246X.2004.02549.x.

[MS95] WF McDonough and SS Sun. The composition of the Earth. *Chem. Geol.*, 120(3):223–253, 1995. URL: http://www.sciencedirect.com/science/article/pii/0009254194001404.

[MA81] JB Minster and DL Anderson. A model of dislocation-controlled rheology for the mantle. *Phil. Trans. R. Soc. Lond.*, 299(1449):319–359, 1981. URL: http://rsta.royalsocietypublishing.org/content/299/1449/319.short.

[MCD+12] I Mosca, L Cobden, A Deuss, J Ritsema, and J Trampert. Seismic and mineralogical structures of the lower mantle from probabilistic tomography. *J. Geophys. Res.: Solid Earth*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2011JB008851/full.

[Mur13] M Murakami. 6 Chemical Composition of the Earth's Lower Mantle: Constraints from Elasticity. In *Physics and Chemistry of the Deep Earth (ed S.-I. Karato)*, pages 183–212. John Wiley & Sons, Ltd, Chichester, UK, 2013. URL: http://books.google.com/books?hl=en\T1\textbackslash{}&lr=\T1\textbackslash{}&id=7z9yES2XNyEC\T1\textbackslash{}&pgis=1.

[MOHH12] M Murakami, Y Ohishi, N Hirao, and K Hirose. A perovskitic lower mantle inferred from high-pressure, high-temperature sound velocity data. *Nature*, 485(7396):90–94, 2012.

[MSH+07] M Murakami, S Sinogeikin, H Hellwig, J Bass, and J Li. Sound velocity of MgSiO3 perovskite to Mbar pressure. *Earth Planet. Sci. Lett.*, 256(1-2):47–54, April 2007. URL: http://www.sciencedirect.com/science/article/pii/S0012821X07000167, doi:10.1016/j.epsl.2007.01.011.

[MOHH09] Motohiko Murakami, Yasuo Ohishi, Naohisa Hirao, and Kei Hirose. Elasticity of MgO to 130 GPa: Implications for lower mantle mineralogy. *Earth Planet. Sci. Lett.*, 277(1-2):123–129, January 2009. URL: http://www.sciencedirect.com/science/article/pii/S0012821X08006675, doi:10.1016/j.epsl.2008.10.010.

[Mur44] F. D. Murnaghan. The compressibility of media under extreme pressures. *Proceedings of the National Academy of Sciences*, 30(9):244–247, 1944. URL: https://www.pnas.org/content/30/9/244, arXiv:https://www.pnas.org/content/30/9/244.full.pdf, doi:10.1073/pnas.30.9.244.

[NTDC12] T Nakagawa, PJ Tackley, F Deschamps, and JAD Connolly. Radial 1-D seismic structures in the deep mantle in mantle convection simulations with self-consistently calculated mineralogy. *Geochem. Geophys. Geosyst.*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2012GC004325/full.

[NFR12] Y Nakajima, DJ Frost, and DC Rubie. Ferrous iron partitioning between magnesium silicate perovskite and ferropericlase and the composition of perovskite in the Earth's lower mantle. *J. Geophys. Res.*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2012JB009151/full.

[NKHO13] M Noguchi, T Komabayashi, K Hirose, and Y Ohishi. High-temperature compression experiments of CaSiO_3 perovskite to lowermost mantle conditions and its thermal equation of state. *Phys. Chem. Miner.*, 40(1):81–91, 2013. URL: http://link.springer.com/article/10.1007/s00269-012-0549-1.

[NOT+11] R Nomura, H Ozawa, S Tateno, K Hirose, J Hernlund, S Muto, H Ishii, and N Hiraoka. Spin crossover and iron-rich silicate melt in the Earth's deep mantle. *Nature*, 473(7346):199–202, 2011. URL: http://www.nature.com/nature/journal/v473/n7346/abs/nature09940.html.

[PR06] M Panning and B Romanowicz. A three-dimensional radially anisotropic model of shear velocity in the whole mantle. *Geophys. J. Int.*, 167(1):361–379, 2006.

[Poi91] JP Poirier. *Introduction to the Physics of the Earth*. Cambridge Univ. Press, Cambridge, England, 1991.

[PH85] R. Powell and T. J. B. Holland. An internally consistent thermodynamic dataset with uncertainties and correlations: 1. methods and a worked example. *Journal of Metamorphic Geology*, 3(4):327–342, 1985. URL: https://doi.org/10.1111/j.1525-1314.1985.tb00324.x, doi:10.1111/j.1525-1314.1985.tb00324.x.

[Pow87] Roger Powell. Darken's quadratic formalism and the thermodynamics of minerals. *American Mineralogist*, 72(1-2):1–11, 1987. URL: http://ammin.geoscienceworld.org/content/72/1-2/1.short.

[PH93] Roger Powell and Tim Holland. On the formulation of simple mixing models for complex phases. *American Mineralogist*, 78(11-12):1174–1180, 1993. URL: http://ammin.geoscienceworld.org/content/78/11-12/1174.short.

[PH99] Roger Powell and Tim Holland. Relating formulations of the thermodynamics of mineral solid solutions; activity modeling of pyroxenes, amphiboles, and micas. *American Mineralogist*, 84(1-2):1–14, 1999. URL: http://ammin.geoscienceworld.org/content/84/1-2/1.abstract, arXiv:http://ammin.geoscienceworld.org/content/84/1-2/1.full.pdf+html.

[RM81] Frank M. Richter and Dan P. McKenzie. Parameterizations for the horizontally averaged temperature of infinite prandtl number convection. *Journal of Geophysical Research: Solid Earth*, 86(B3):1738–1744, 1981. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB086iB03p01738, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JB086iB03p01738, doi:https://doi.org/10.1029/JB086iB03p01738.

[RDvHW11] J Ritsema, A Deuss, H. J. van Heijst, and J.H. Woodhouse. S40RTS: a degree-40 shear-velocity model for the mantle from new Rayleigh wave dispersion, teleseismic traveltime and normal-mode splitting function. *Geophys. J. Int.*, 184(3):1223–1236, 2011. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2010.04884.x/full.

[SE15] S. Saxena and G. Eriksson. Thermodynamics of Fe–S at ultra-high pressure. *Calphad*, 51:202–205, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0364591615300249, doi:https://doi.org/10.1016/j.calphad.2015.09.009.

[Sch16] F. A. H. Schreinemakers. In-, mono-, and di-variant equilibria. VIII. Further consideration of the bivariant regions; the turning lines. *Proc. K. Akad. Wet. (Netherlands)*, 18:1539–1552, 1916.

[SZN12] BSA Schuberth, C Zaroli, and G Nolet. Synthetic seismograms for a synthetic Earth: long-period P- and S-wave traveltime variations can be explained by temperature alone. *Geophys. J. Int.*, 188(3):1393–1412, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2011.05333.x/full.

[SFBG10] NA Simmons, AM Forte, L Boschi, and SP Grand. GyPSuM: A joint tomographic model of mantle density and seismic wave speeds. *J. Geophys. Res.*, 115(B12):B12310, 2010. URL: http://www.agu.org/pubs/crossref/2010/2010JB007631.shtml.

[SMJM12] NA Simmons, SC Myers, G Johanneson, and E Matzel. LLNL-G3Dv3: Global P wave tomography model for improved regional and teleseismic travel time prediction. *J. Geophys. Res.*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2012JB009525/full.

[SD04] F.D. Stacey and P.M. Davis. High pressure equations of state with applications to the lower mantle and core. *Physics of the Earth and Planetary Interiors*, 142(3-4):137–184, may 2004. URL: http://linkinghub.elsevier.com/retrieve/pii/S0031920104001049, doi:10.1016/j.pepi.2004.02.003.

[Sta77] Frank D. Stacey. A thermal model of the Earth. *Physics of the Earth and Planetary Interiors*, 15(4):341–348, 1977. URL: https://www.sciencedirect.com/science/article/pii/0031920177900966, doi:https://doi.org/10.1016/0031-9201(77)90096-6.

[SD00] Frank D. Stacey and Frank D. The K-primed approach to high-pressure equations of state. *Geophysical Journal International*, 143(3):621–628, dec 2000. URL: https://academic.oup.com/gji/article-lookup/doi/10.1046/j.1365-246X.2000.00253.x, doi:10.1046/j.1365-246X.2000.00253.x.

[SLB05] L Stixrude and C Lithgow-Bertelloni. Thermodynamics of mantle minerals–I. Physical properties. *Geophys. J. Int.*, 162(2):610–632, 2005. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2005.02642.x/full.

[SLB11] L Stixrude and C Lithgow-Bertelloni. Thermodynamics of mantle minerals–II. Phase equilibria. *Geophys. J. Int.*, 184(3):1180–1213, 2011. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2010.04890.x/full.

[SLB12] L Stixrude and C Lithgow-Bertelloni. Geophysics of chemical heterogeneity in the mantle. *Annu. Rev. Earth Planet. Sci.*, 40:569–595, 2012. URL: http://www.annualreviews.org/doi/abs/10.1146/annurev.earth.36.031207.124244.

[SDG11] Elinor Styles, D. Rhodri Davies, and Saskia Goes. Mapping spherical seismic into physical structure: biases from 3-D phase-transition and thermal boundary-layer heterogeneity. *Geophys. J. Int.*, 184(3):1371–1378, March 2011. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2010.04914.x, doi:10.1111/j.1365-246X.2010.04914.x.

[Sun91] B. Sundman. An assessment of the fe-o system. *Journal of Phase Equilibria*, 12(2):127–140, 1991. URL: https://doi.org/10.1007/BF02645709, doi:10.1007/BF02645709.

[Tac00] PJ Tackley. Mantle convection and plate tectonics: Toward an integrated physical and chemical theory. *Science*, 288(5473):2002–2007, 2000. URL: http://www.sciencemag.org/content/288/5473/2002.short.

[Tem45] Mo Temkin. Mixtures of fused salts as ionic solutions. *Acta Phys. Chem., USSR*, 20:411, 1945.

[TRCT05] A To, B Romanowicz, Y Capdeville, and N Takeuchi. 3D effects of sharp boundaries at the borders of the African and Pacific Superplumes: Observation and modeling. *Earth Planet. Sci. Lett.*, 233(1-2):1447–1460, 2005.

[TDRY04] Jeannot Trampert, Frédéric Deschamps, Joseph Resovsky, and Dave Yuen. Probabilistic tomography maps chemical heterogeneities throughout the lower mantle. *Science (New York,*

N.Y.), 306(5697):853–6, October 2004. URL: http://www.sciencemag.org/content/306/5697/853.full, doi:10.1126/science.1101996.

[TVV01] Jeannot Trampert, Pierre Vacher, and Nico Vlaar. Sensitivities of seismic velocities to temperature, pressure and composition in the lower mantle. *Phys. Earth Planet. Int.*, 124(3-4):255–267, August 2001. URL: http://www.sciencedirect.com/science/article/pii/S0031920101002011, doi:10.1016/S0031-9201(01)00201-1.

[VSFR87] Pascal Vinet, John R Smith, John Ferrante, and James H Rose. Temperature effects on the universal equation of state of solids. *Physical Review B*, 35(4):1945, 1987. doi:10.1103/PhysRevB.35.1945.

[VFSR86] PJJR Vinet, J Ferrante, JR Smith, and JH Rose. A universal equation of state for solids. *Journal of Physics C: Solid State Physics*, 19(20):L467, 1986. doi:10.1088/0022-3719/19/20/001.

[WB07] E. Bruce Watson and Ethan F. Baxter. Diffusion in solid-earth systems. *Earth and Planetary Science Letters*, 253(3–4):307 – 327, 2007. URL: http://www.sciencedirect.com/science/article/pii/S0012821X06008168, doi:https://doi.org/10.1016/j.epsl.2006.11.015.

[WDOConnell76] JP Watt, GF Davies, and RJ O'Connell. The elastic properties of composite materials. *Rev. Geophys.*, 14(4):541–563, 1976. URL: http://onlinelibrary.wiley.com/doi/10.1029/RG014i004p00541/full.

[WPB08] R. W. White, R. Powell, and J. A. Baldwin. Calculated phase equilibria involving chemical potentials to investigate the textural evolution of metamorphic rocks. *Journal of Metamorphic Geology*, 26(2):181–198, 2008. URL: https://doi.org/10.1111/j.1525-1314.2008.00764.x, doi:10.1111/j.1525-1314.2008.00764.x.

[WP11] RW White and R Powell. On the interpretation of retrograde reaction textures in granulite facies rocks. *Journal of Metamorphic Geology*, 29(1):131–149, 2011.

[WJW13] Z Wu, JF Justo, and RM Wentzcovitch. Elastic Anomalies in a Spin-Crossover System: Ferropericlase at Lower Mantle Conditions. *Phys. Rev. Lett.*, 110(22):228501, 2013. URL: http://prl.aps.org/abstract/PRL/v110/i22/e228501.

[ZSB13] Zhigang Zhang, Lars Stixrude, and John Brodholt. Elastic properties of MgSiO3-perovskite under lower mantle conditions and the composition of the deep Earth. *Earth Planet. Sci. Lett.*, 379:1–12, October 2013. URL: http://www.sciencedirect.com/science/article/pii/S0012821X13004093, doi:10.1016/j.epsl.2013.07.034.

[AndersonCrerar89] G. M. Anderson and D. A. Crerar. *Thermodynamics in geochemistry: The equilibrium model*. Oxford University Press, 1989.

[AndersonAhrens94] W. W. Anderson and T. J. Ahrens. An equation of state for liquid iron and implications for the Earth's core. *Journal of Geophysical Research*, 99:4273–4284, March 1994. doi:10.1029/93JB03158.

[Darken67] L. S. Darken. Thermodynamics of binary metallic solutions. *Metallurgical Society of AIME Transactions*, 239:80–89, 1967.

[deKokerKarkiStixrude13] N. de Koker, B. B. Karki, and L. Stixrude. Thermodynamics of the MgO-SiO$_2$ liquid system in Earth's lowermost mantle from first principles. *Earth and Planetary Science Letters*, 361:58–63, January 2013. doi:10.1016/j.epsl.2012.11.026.

[HamaSuito98] J. Hama and K. Suito. High-temperature equation of state of CaSiO $_3$ perovskite and its implications for the lower mantle. *Physics of the Earth and Planetary Interiors*, 105:33–46, February 1998. doi:10.1016/S0031-9201(97)00074-5.

[HollandPowell03] T. Holland and R. Powell. Activity-composition relations for phases in petrological calculations: an asymmetric multicomponent formulation. *Contributions to Mineralogy and Petrology*, 145:492–501, 2003. doi:10.1007/s00410-003-0464-z.

[HollandPowell98] T. J. B. Holland and R. Powell. An internally consistent thermodynamic data set for phases of petrological interest. *Journal of Metamorphic Geology*, 16(3):309–343, 1998. URL: https://doi.org/10.1111/j.1525-1314.1998.00140.x, doi:10.1111/j.1525-1314.1998.00140.x.

[HollandPowell11] T. J. B. Holland and R. Powell. An improved and extended internally consistent thermodynamic dataset for phases of petrological interest, involving a new equation of state for solids. *Journal of Metamorphic Geology*, 29(3):333–383, 2011. URL: https://doi.org/10.1111/j.1525-1314.2010.00923.x, doi:10.1111/j.1525-1314.2010.00923.x.

[HuangChow74] Y. K. Huang and C. Y. Chow. The generalized compressibility equation of Tait for dense matter. *Journal of Physics D Applied Physics*, 7:2021–2023, October 1974. doi:10.1088/0022-3727/7/15/305.

[Myhill18] R. Myhill. The elastic solid solution model for minerals at high pressures and temperatures. *Contributions to Mineralogy and Petrology*, 173(2):12, February 2018. doi:10.1007/s00410-017-1436-z.

[Myhill22] R. Myhill. An anisotropic equation of state for high pressure, high temperature applications. *Geophysical Journal International*, May 2022. doi:10.1093/gji/ggac180.

[NissenMeyervanDrielStahler+14] T. Nissen-Meyer, M. van Driel, S. C. Stähler, K. Hosseini, S. Hempel, L. Auer, A. Colombi, and A. Fournier. AxiSEM: broadband 3-D seismic wavefields in axisymmetric media. *Solid Earth*, 5:425–445, June 2014. doi:10.5194/se-5-425-2014.

[Putnis92] A. Putnis. *An Introduction to Mineral Sciences*. Cambridge University Press, November 1992.

[Rydberg32] R. Rydberg. Graphische Darstellung einiger bandenspektroskopischer Ergebnisse. *Zeitschrift fur Physik*, 73:376–385, May 1932. doi:10.1007/BF01341146.

[StaceyBrennanIrvine81] F. D. Stacey, B. J. Brennan, and R. D. Irvine. Finite strain theories and comparisons with seismological data. *Geophysical Surveys*, 4:189–232, April 1981. doi:10.1007/BF01449185.

[vanLaar06] J. J. van Laar. Sechs vorträge über das thermodynamischer potential. *Vieweg, Brunswick*, 1906.

INDEX

A

B

C

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

Z