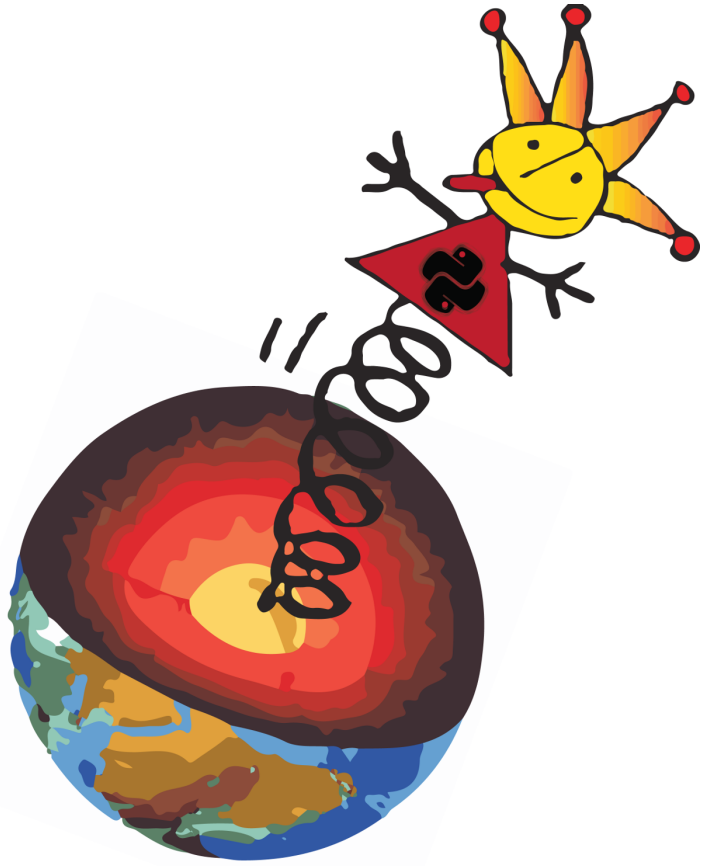
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

BurnMan

a thermodynamics and thermoelasticity toolkit

User Manual

Version 1.0.1



Robert Myhill
Sanne Cottaar
Timo Heister
Ian Rose
Cayman Unterborn

http://geodynamics.org

CONTENTS

CHAPTER

ONE

INTRODUCING BURNMAN 1.0.1

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, solid 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

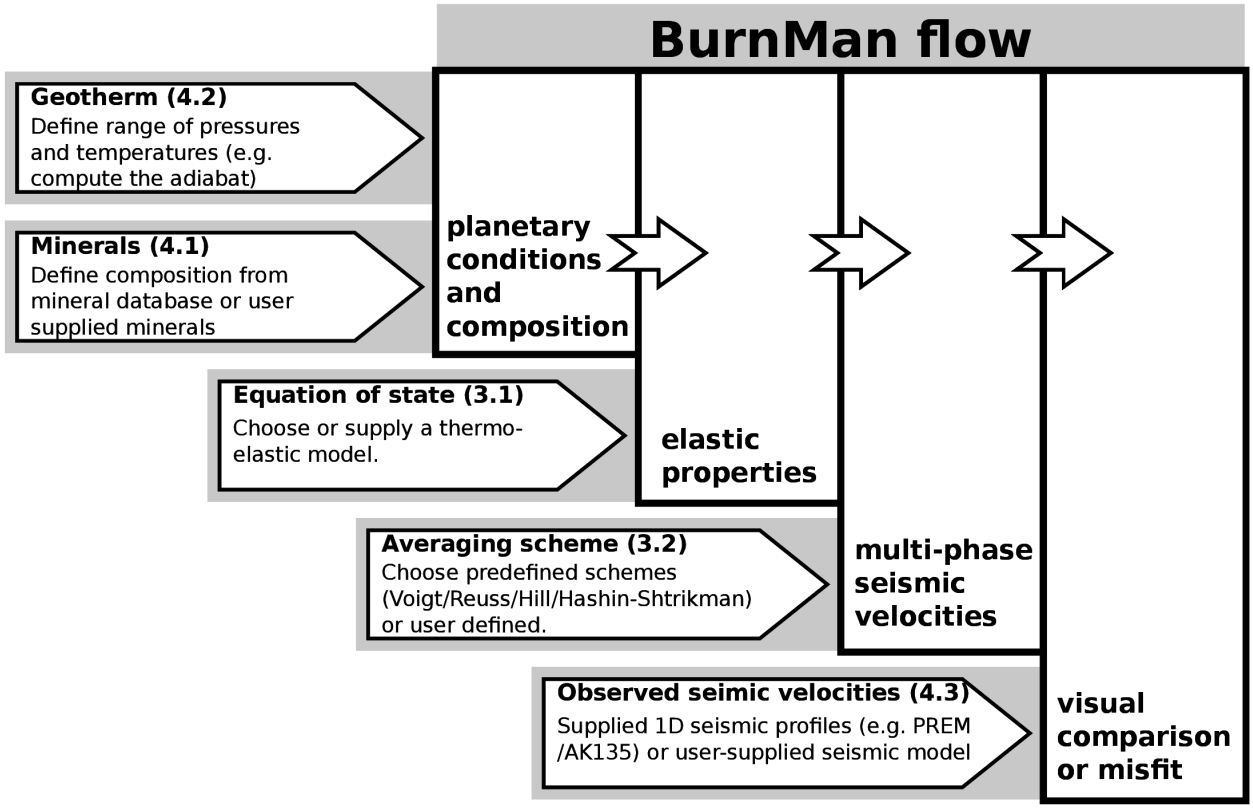
- an extensive suite of unit tests to ensure code functions as intended
- a series of benchmarks comparing BurnMan output with published data
- a directory containing user-contributed code from published papers

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, solid 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 Installation

1.3.1 Requirements

- Python 3.7+
- Python modules: NumPy, SciPy, SymPy, Matplotlib

1.3.2 Source code

The source code can be found at https://github.com/geodynamics/burnman.

1.3.3 Install under Ubuntu

1. Install dependencies using apt by opening a terminal window and entering `sudo apt-get install python python-scipy python-numpy python-sympy python-matplotlib git`
2. Clone the BurnMan repository `git clone https://github.com/geodynamics/burnman.git`
3. Go to the Burnman examples directory and type: `python example_beginner.py` Figures should show up, indicating that it is working.

1.3.4 Install on a Mac

1. get Xcode
2. If you don't have Python yet, download it (for free) from python.org/download . Make sure to use Python 3.7+. To check your version of python, type the following in a terminal: `python --version`
3. Install the latest Numpy version from http://sourceforge.net/projects/numpy/files/NumPy/
4. Install the latest Scipy from http://sourceforge.net/projects/scipy/files/
5. Install the latest Sympy from http://sourceforge.net/projects/sympy/files/
6. Install the latest Matplotlib from http://sourceforge.net/projects/matplotlib/files/matplotlib/matplotlib-1.1.1/
7. Clone the BurnMan repository `git clone https://github.com/geodynamics/burnman.git`
8. Go to the Burnman examples directory and type `python example_beginner.py` Figures should show up, indicating that it is working.

1.3.5 Install under Windows

To get Python running under Windows:

1. Download Python from http://www.python.org/ and install
2. Go to http://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy, download and install
3. Go to http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy, download and install
4. Go to http://www.lfd.uci.edu/~gohlke/pythonlibs/#sympy, download and install
5. Go to http://www.lfd.uci.edu/~gohlke/pythonlibs/#matplotlib, download and install
6. Download BurnMan from github (https://github.com/geodynamics/burnman)
7. Open Python Shell (IDLE Python GUI)
8. File – Open – find one of the example files
9. Run the module (or press F5)

1.4 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. (2021): BurnMan v1.0.1 [Software]. Computational Infrastructure for Geodynamics. Zenodo. (https://doi.org/10.5281/zenodo.5552756)
- 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)

1.5 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.

1.6 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).

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

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.

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

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 essentially a symmetric solid solution model between endmembers, with an excess configurational entropy term predicted on the basis of 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 Solid Solution Properties

Many minerals can exist over a finite region of composition space. These spaces are bounded by endmembers (which may themselves not be stable), and each individual mineral can then be described as a solid solution of those endmembers. At an atomic level, 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 solid solution calculations. The chemical formula of many low pressure garnets exist within the solid solution:

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

We typically calculate solid solution properties by appropriate differentiation of the Gibbs Free 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 solid solutions

A solid solution is not simply a mechanical mixture of its constituent endmembers. The mixing of different elements on sites results in an excess configurational entropy

$$\mathcal{S}_{\text{conf}} = R \ln \prod_s (X_c^s)^\nu$$

where s is a site in the lattice M, c are the cations mixing on site s and ν is the number of s sites in the formula unit. Solid solutions where this configurational entropy is the only deviation from a mechanical mixture are termed *ideal*. From this expression, we can see that

$$\alpha_{\text{ideal},i} = \prod_s (X_c^s)^\nu$$

2.2.1.2 Symmetric solid solutions

Many real minerals are not well approximated as ideal solid solutions. Deviations are the result of elastic and chemical interactions between ions with different physical and chemical characteristics. Regular (symmetric) solid solution models are designed to account for the simplest form of deviations from ideality, by allowing the addition of excess enthalpies, non-configurational entropies and volumes 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 solid solutions

Some solid solutions exhibit asymmetric excess terms. These can be accounted for with an asymmetric solid 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 solid 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.

2.2.2 Thermodynamic and thermoelastic properties

From the preceeding equations, we can define the thermodynamic potentials of solid 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 solid 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 solid solution.

2.2.4 Including spin transitions

The regular solid 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 ($\Delta \upsilon^0$) for the Fe-Mg exchange between perovskite and ferropericlase

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

where R is the gas constant and P_0 the reference pressure. As a default, we adopt the average $\Delta \upsilon^0$ 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 ($MgSiO_3$), fe_perovskite ($FeSiO_3$), 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 $(Mg, Fe)SiO_3$ and $(Mg, Fe)O$ 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, 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

TUTORIAL

The tutorial for BurnMan currently consists of three separate units:

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

3.1 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.

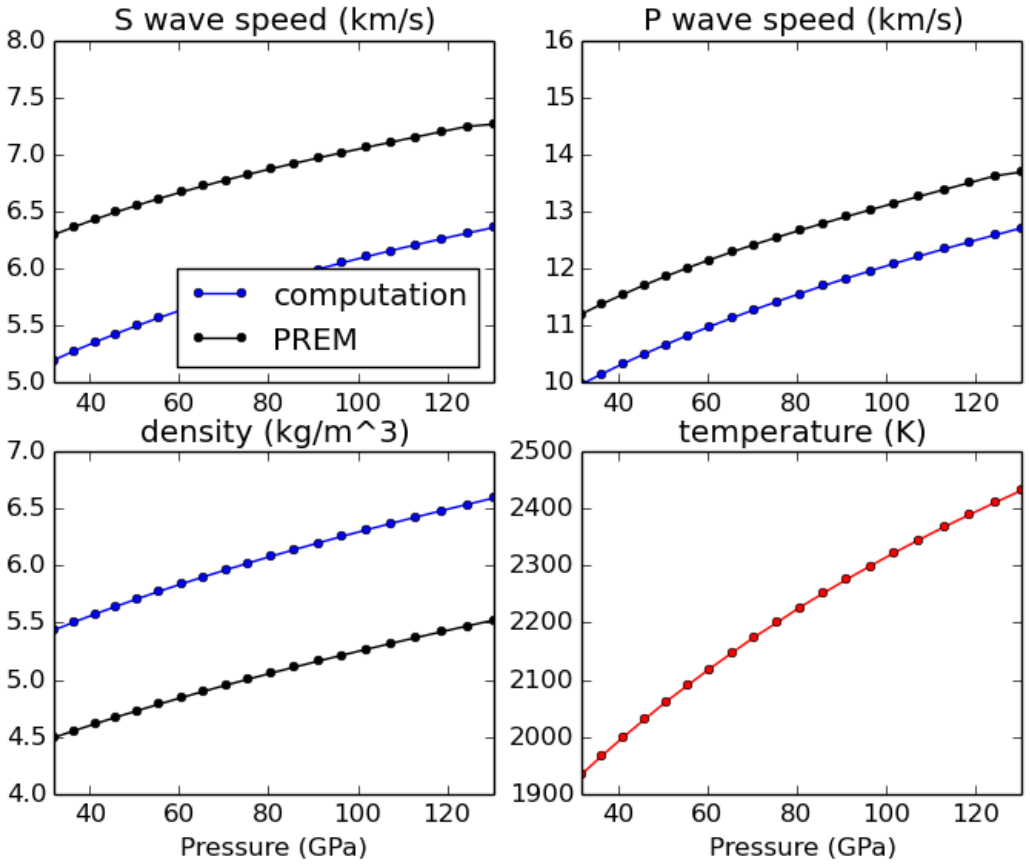
3) Plot those elastic properties, and compare them to a seismic model, in this case PREM

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…

3.2 CIDER 2014 BurnMan Tutorial — step 2

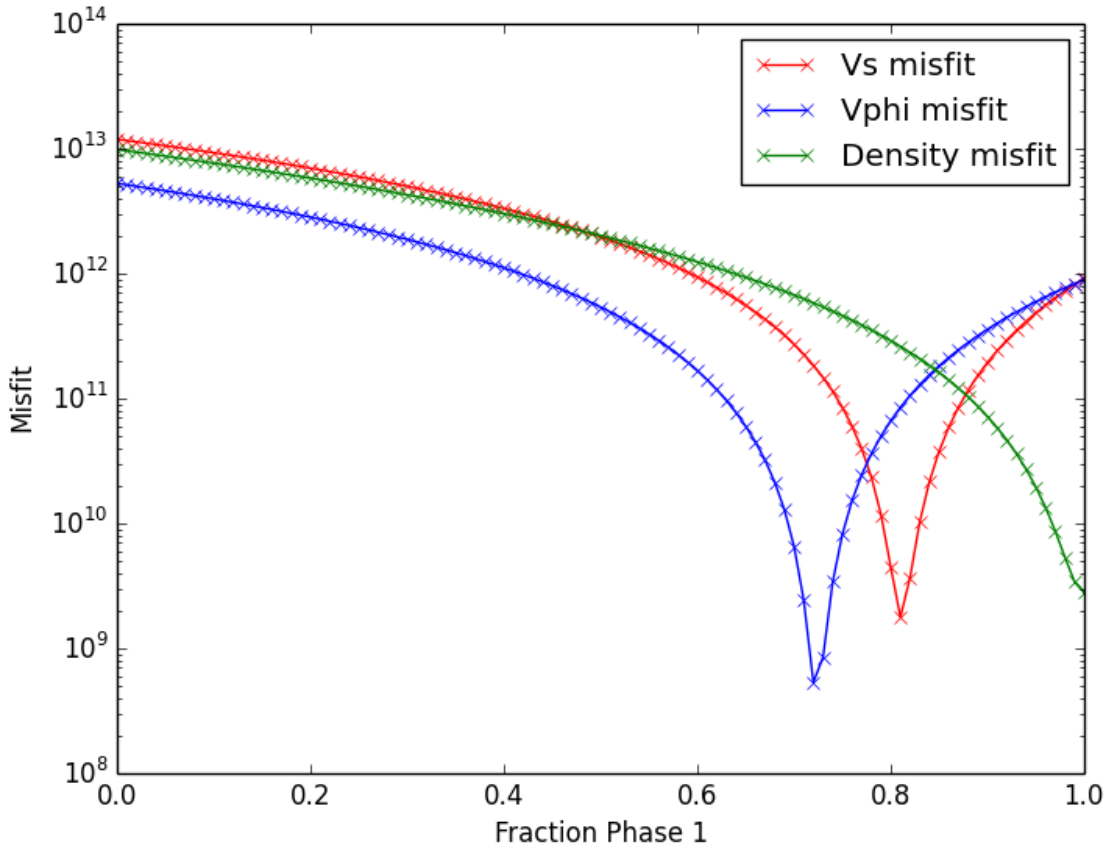
In this second part of the tutorial we try to get a closer fit to our 1D seismic reference model. In the simple Mg, Si, and O model that we used in step 1 there was one free parameter, namely phase_1_fraction, which goes between zero and one.

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:



3.3 CIDER 2014 BurnMan Tutorial — step 3

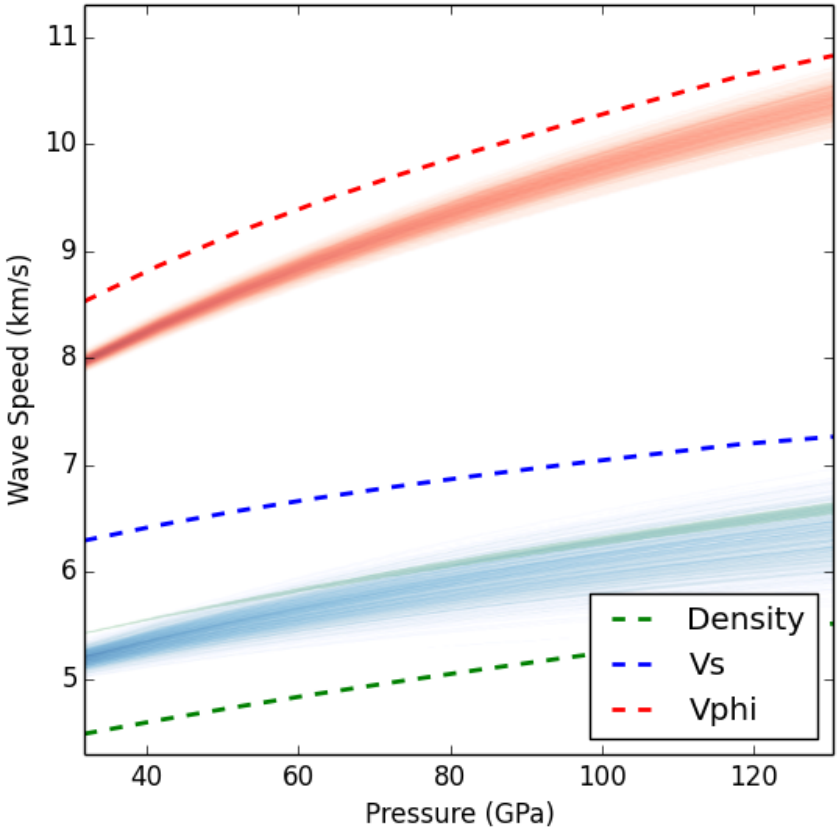
In the previous two steps of the tutorial we tried to find a very simple mineralogical model that best fit the 1D seismic model PREM. But we know that there is consideral uncertainty in many of the mineral physical parameters that control how the elastic properties of minerals change with pressure and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

FOUR

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*. We suggest starting with *Class examples*, which introduces the main class types in BurnMan. The *Tutorial* then provides a useful introduction to the seismic tools for new users of BurnMan.

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)*.

4.1 Class examples

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

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

4.1.1 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*).

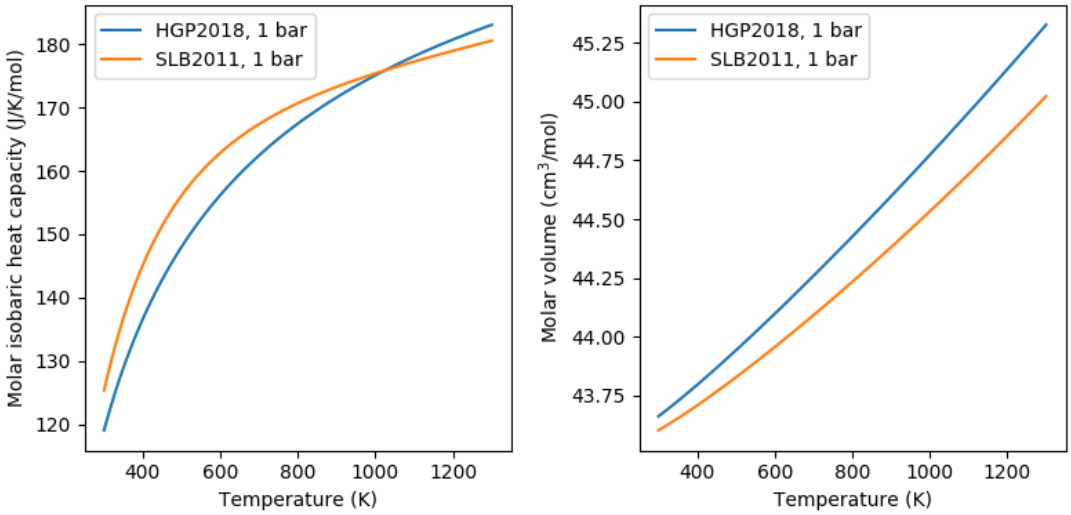
Uses:

- *Mineral databases*
- `burnman.Mineral`

Demonstrates:

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

Resulting figure:



4.1.2 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))

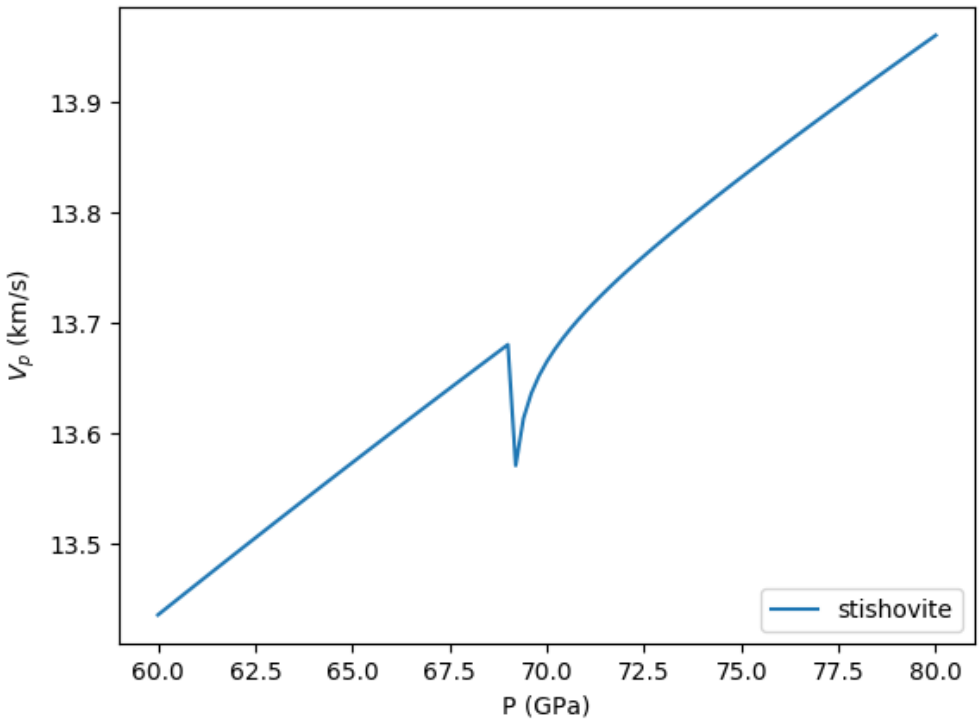
Uses:

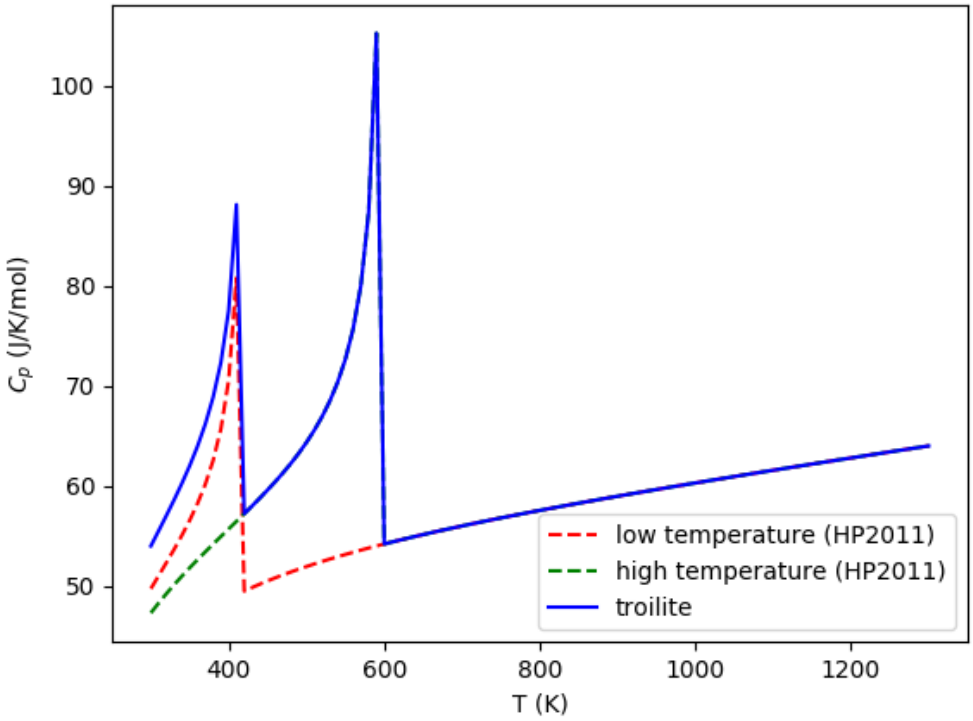
- *Mineral databases*

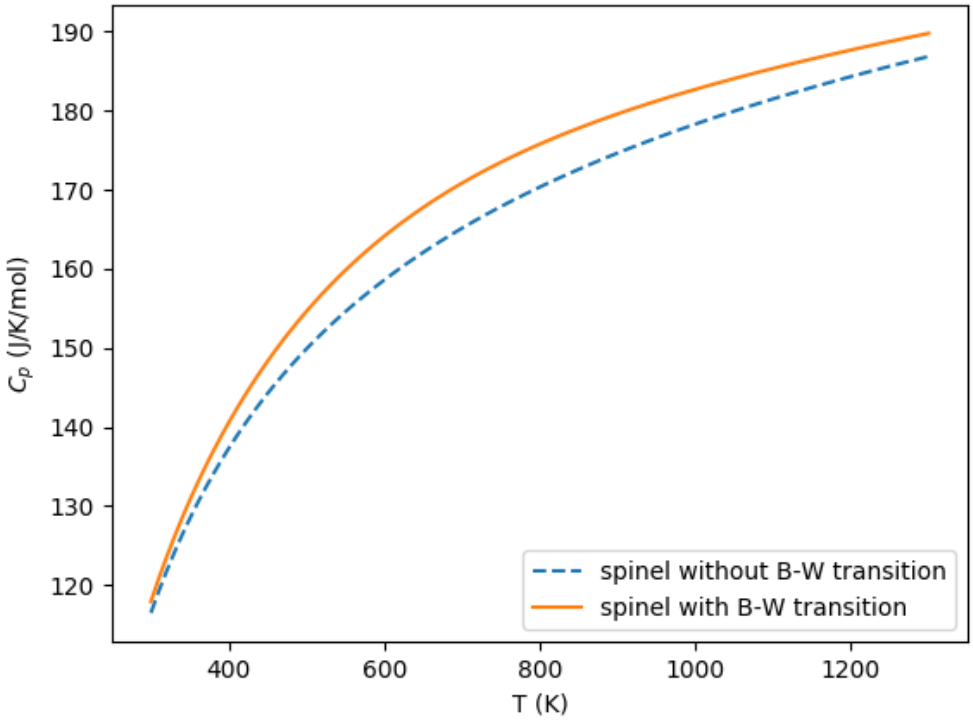
Demonstrates:

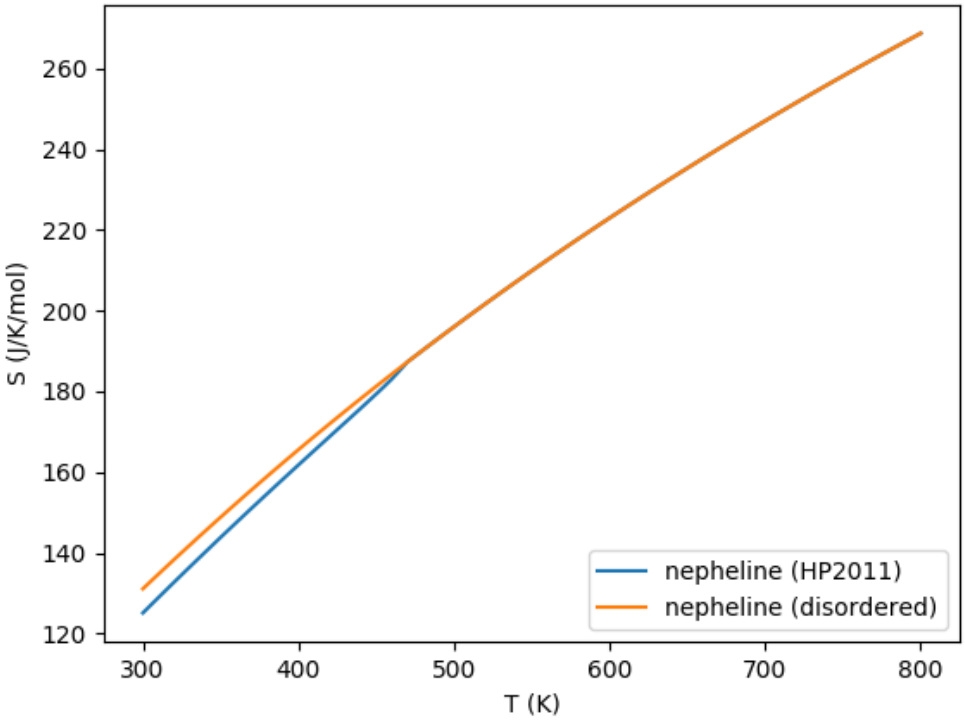
- creating a mineral with excess contributions
- calculating thermodynamic properties

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



4.1.3 example_solid_solution

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

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

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

These solid 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:

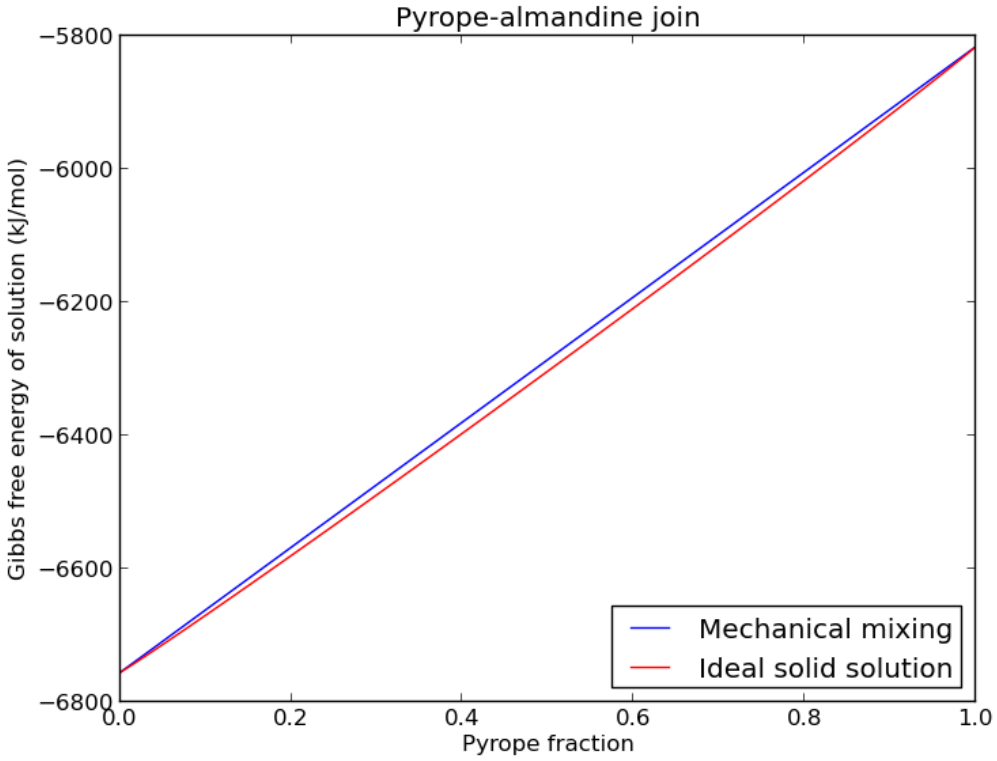
- *Mineral databases*
- `burnman.SolidSolution`

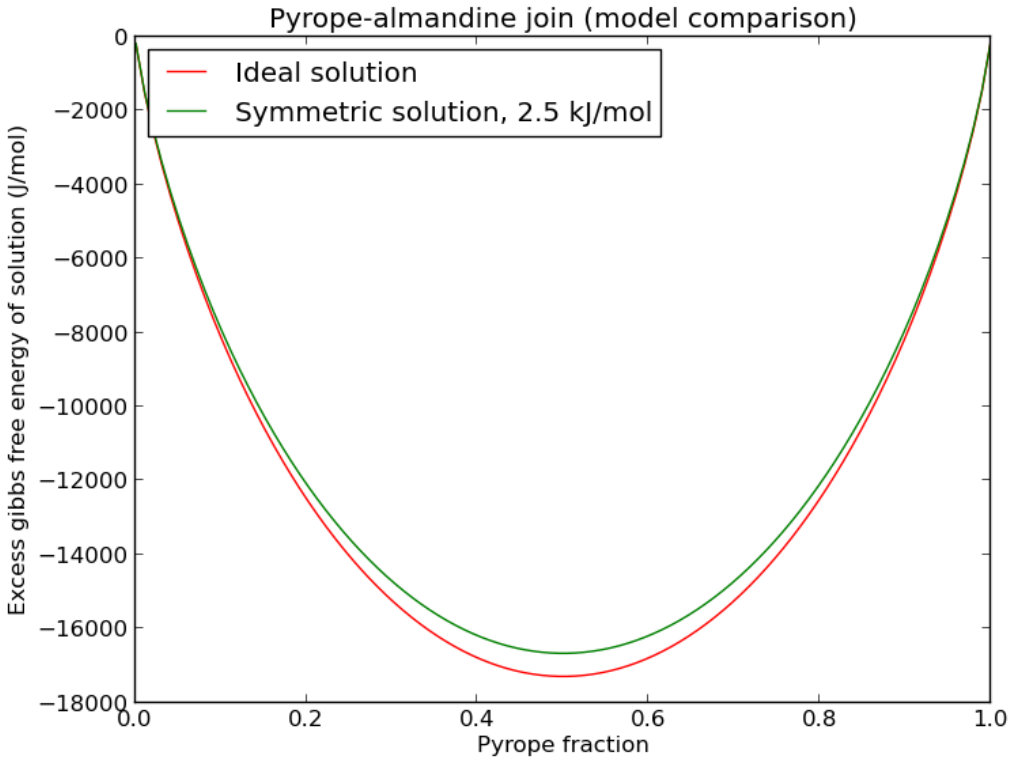
- *burnman.SolutionModel*

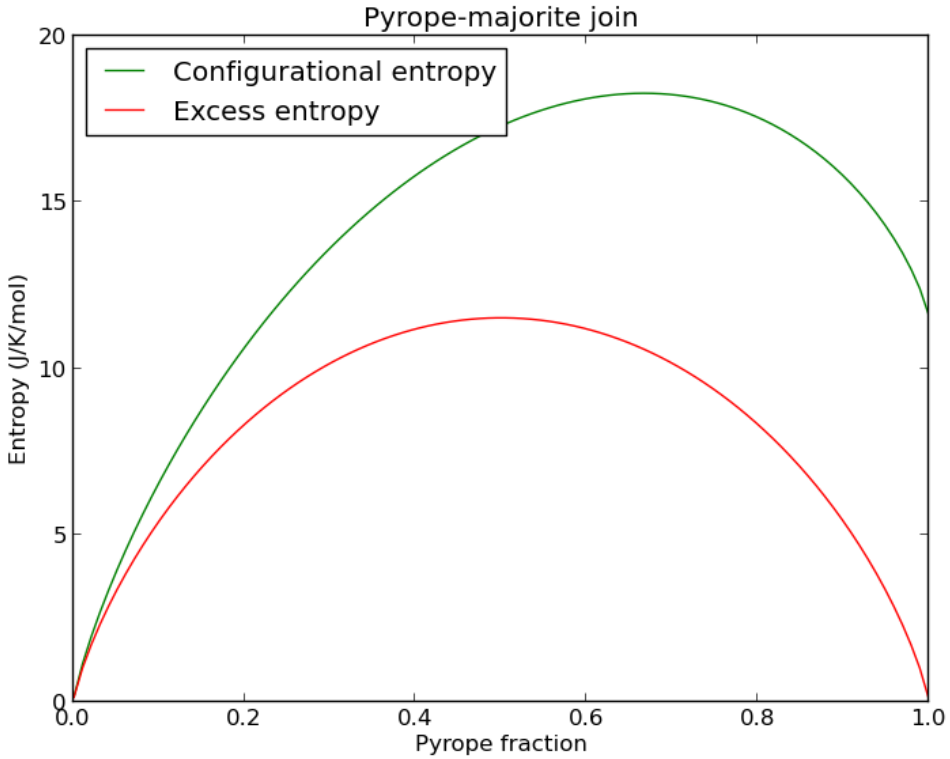
Demonstrates:

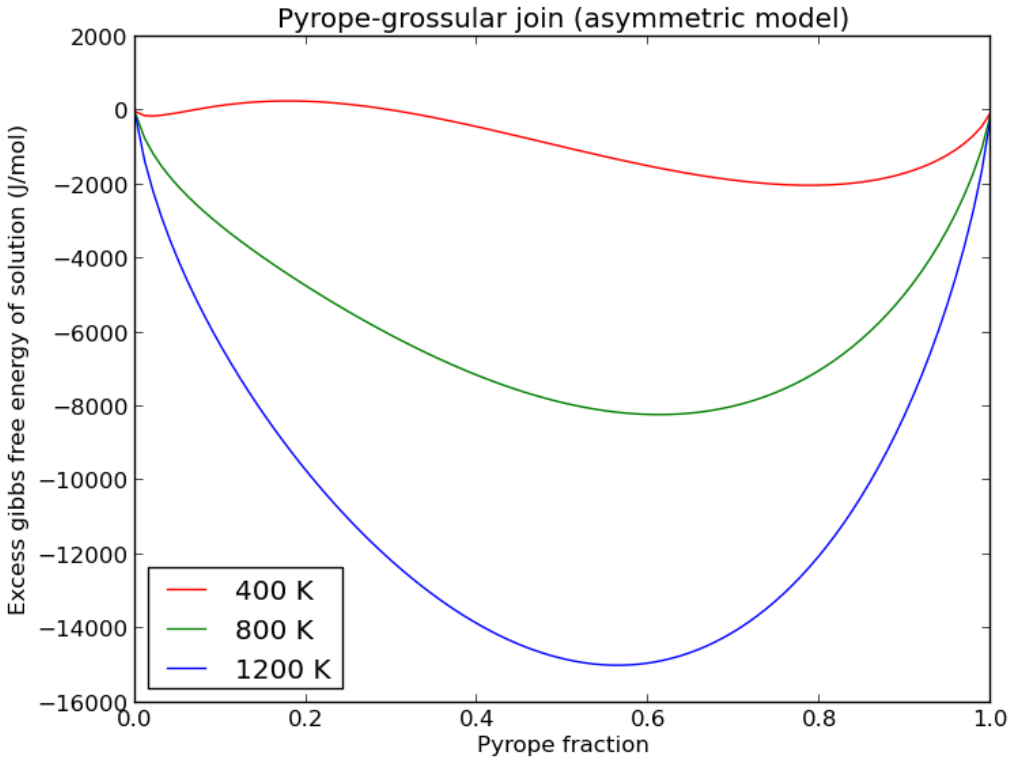
- Different ways to define a solid solution
- How to set composition and state
- How to output thermodynamic and thermoelastic properties

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



4.1.4 example_composite

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

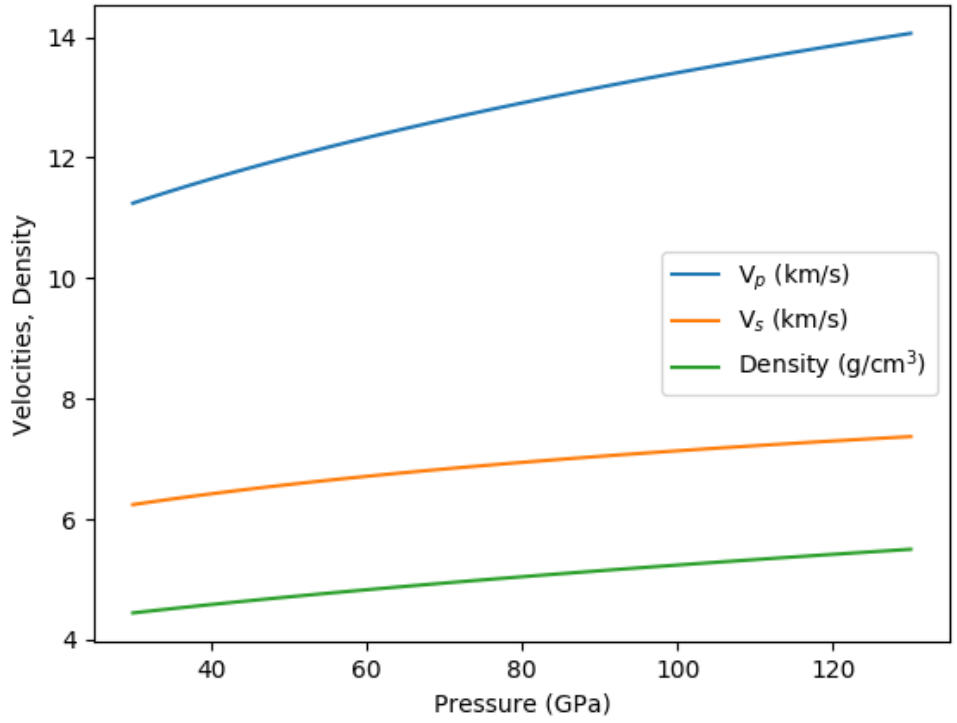
Uses:

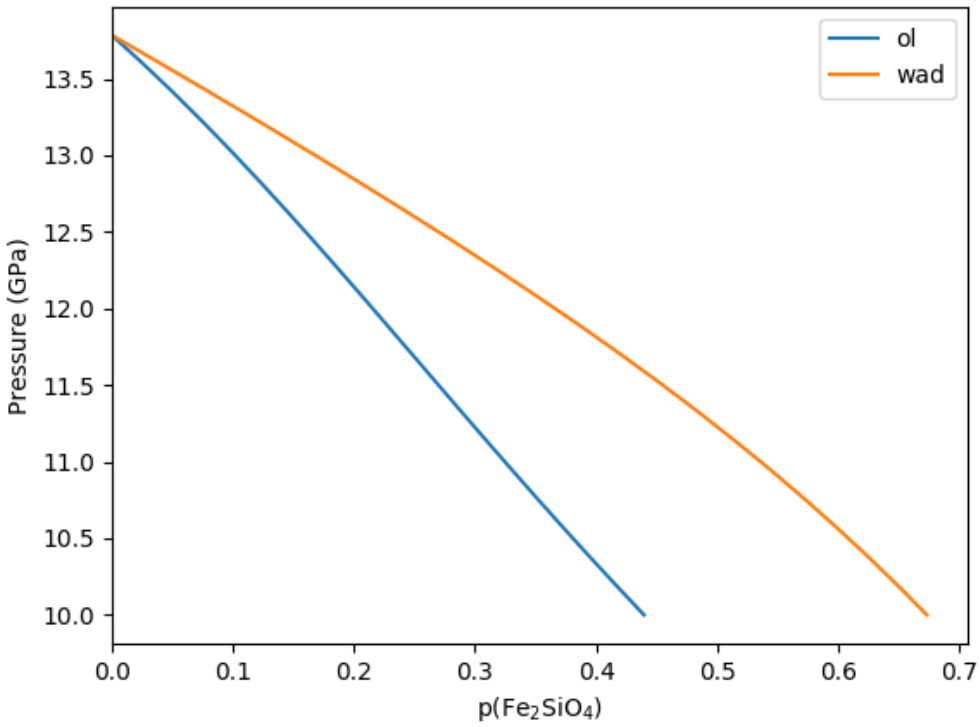
- *Mineral databases*
- `burnman.Mineral`
- `burnman.SolidSolution`
- `burnman.Composite`

Demonstrates:

- How to initialize a composite object containing minerals and solid 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

4.1.5 example_anisotropy

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

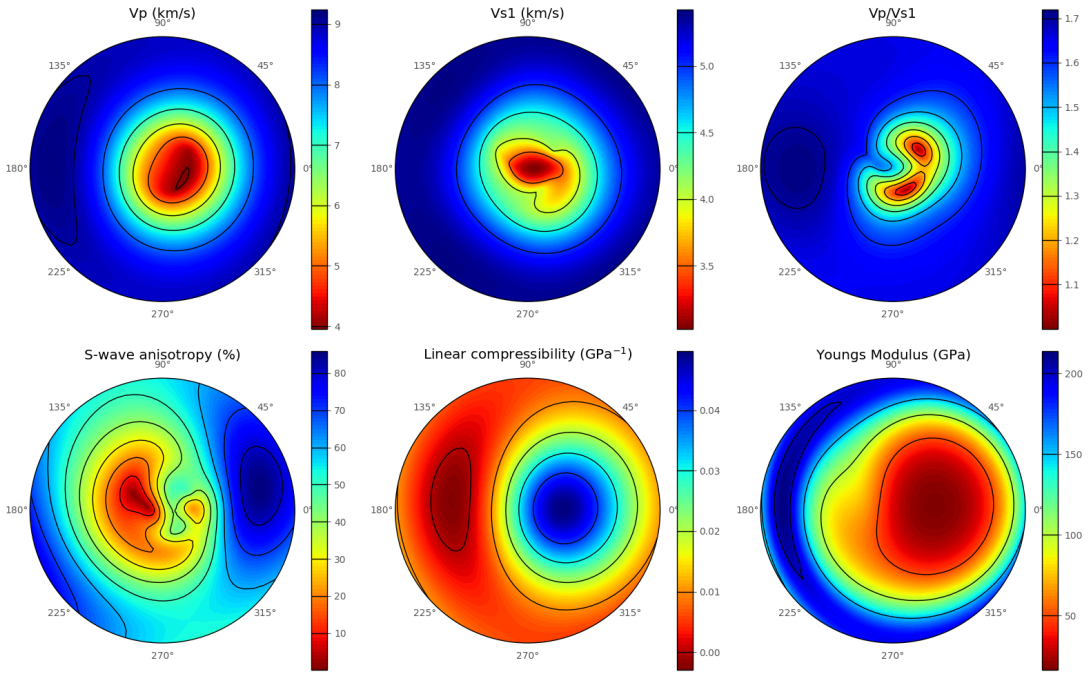
Specifically uses:

- *burnman.AnisotropicMaterial*

Demonstrates:

- anisotropic functions

Resulting figure:



4.1.6 example_anisotropic_mineral

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

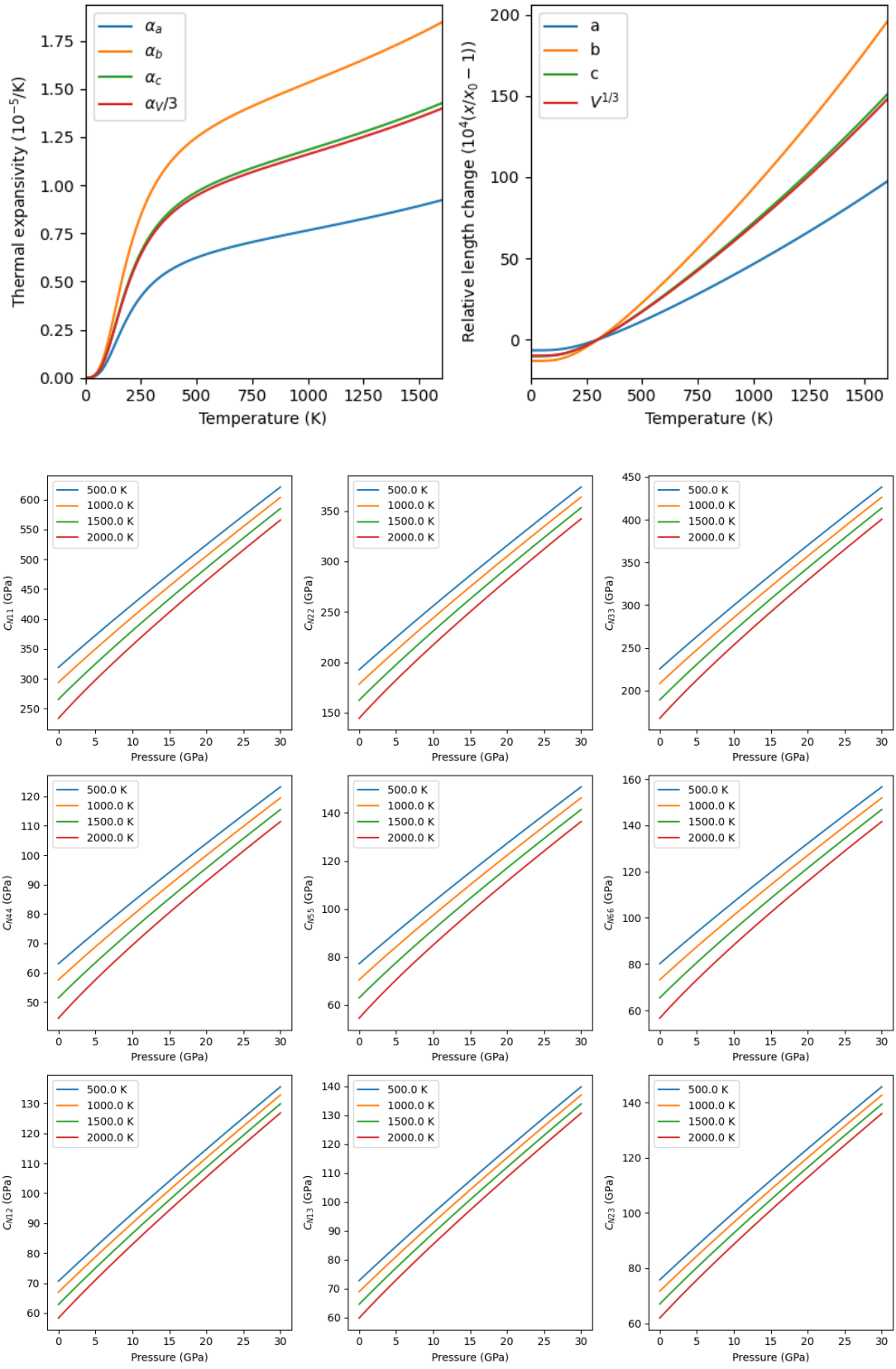
Specifically uses:

- *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)

4.1.7 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

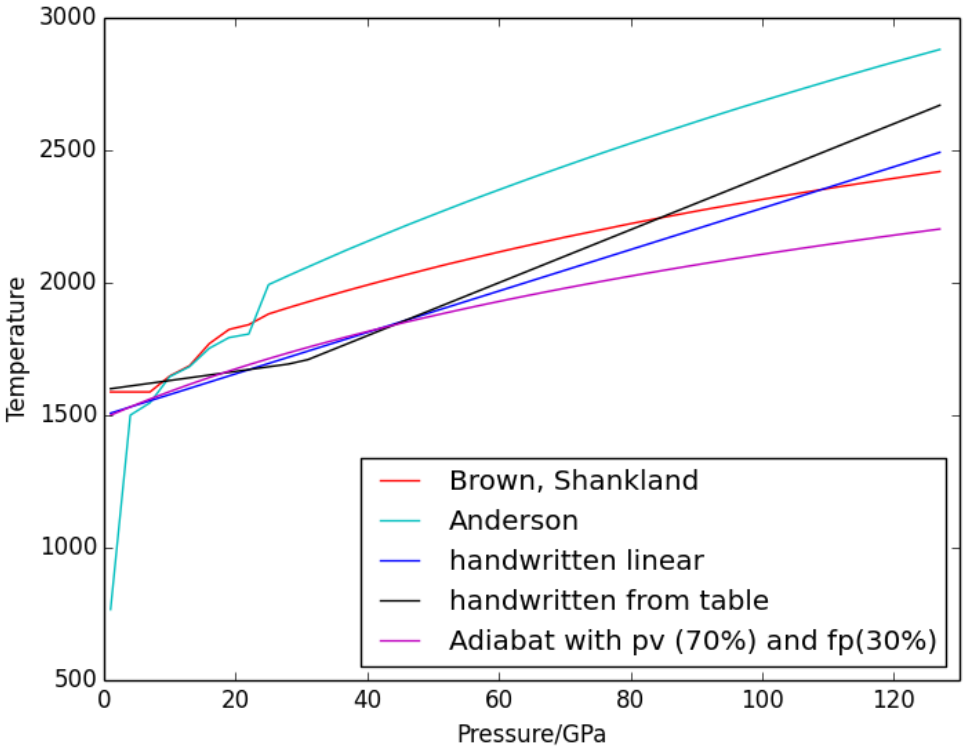
Uses:

- `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:



4.1.8 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

4.2 Simple Examples

The following is a list of simple examples:

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

4.2.1 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.

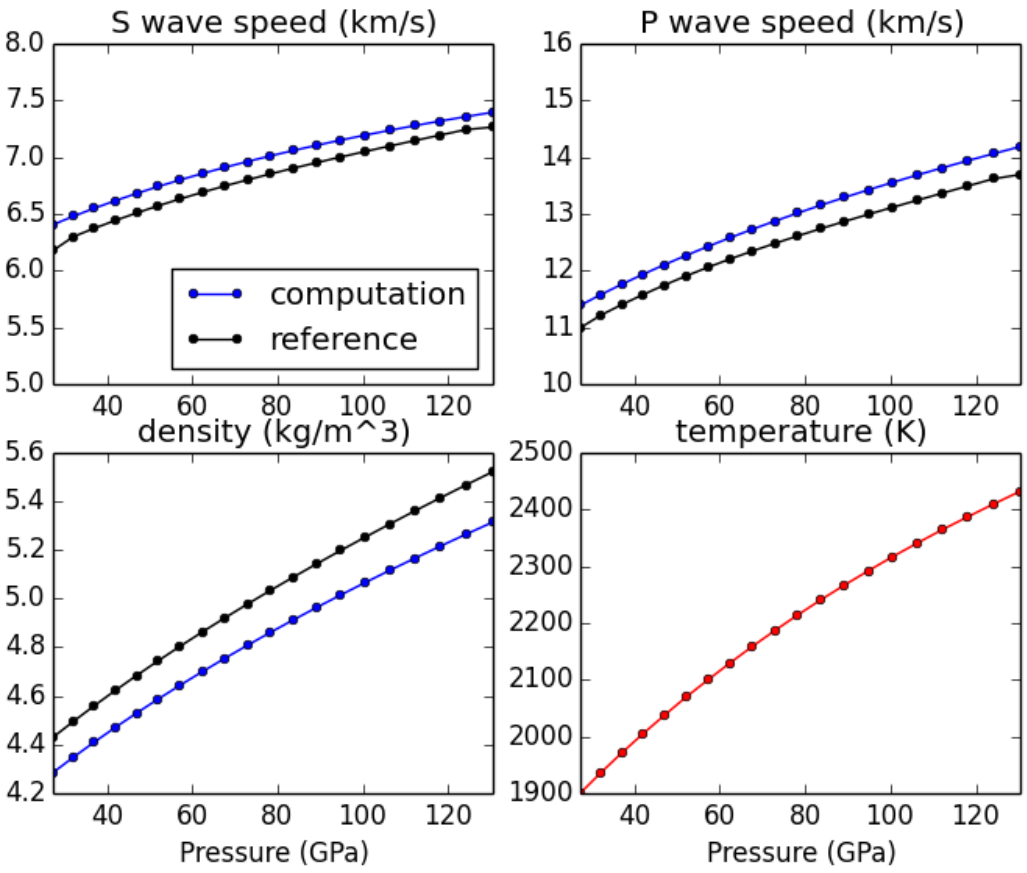
Uses:

- *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:



4.2.2 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].

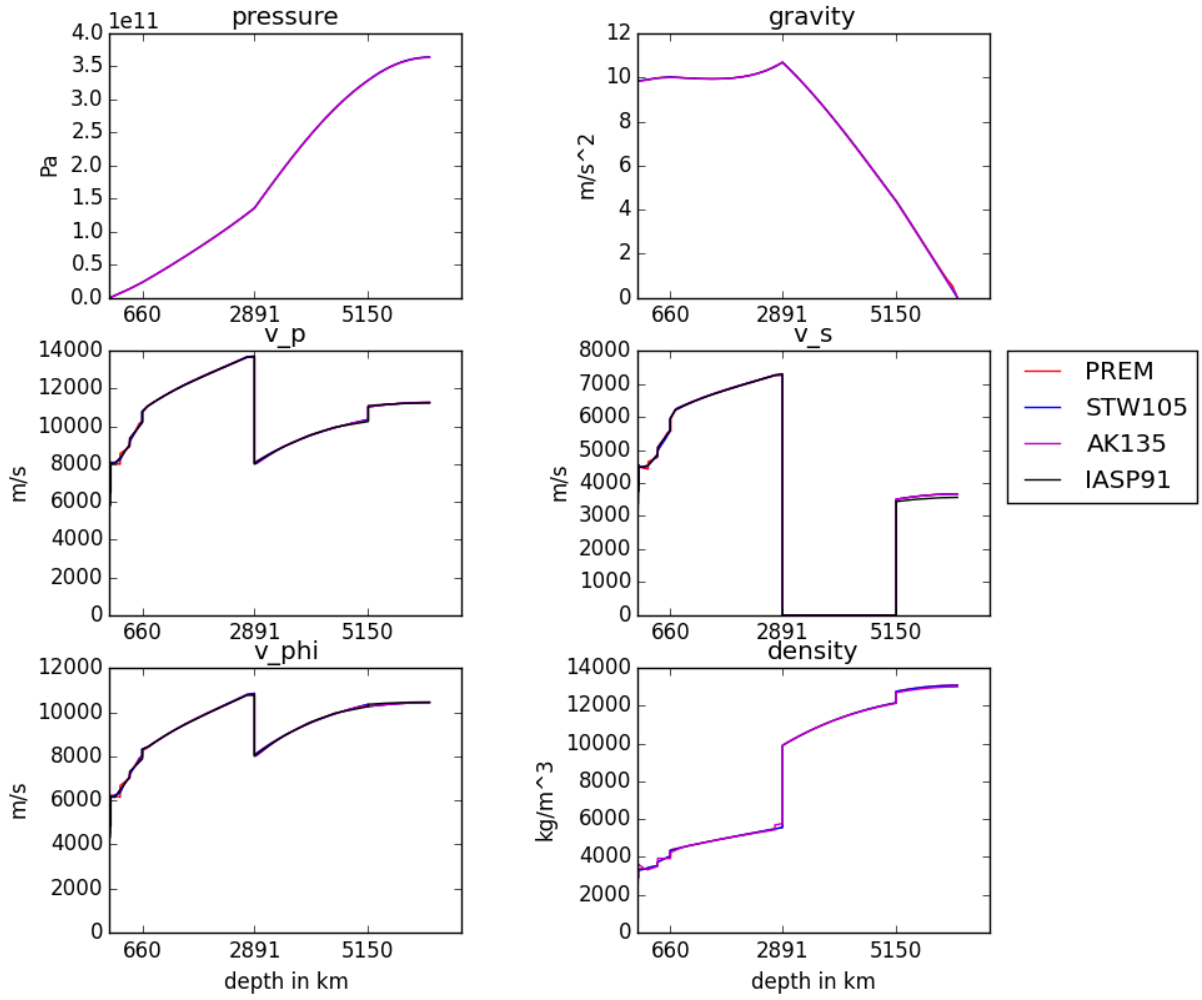
Uses:

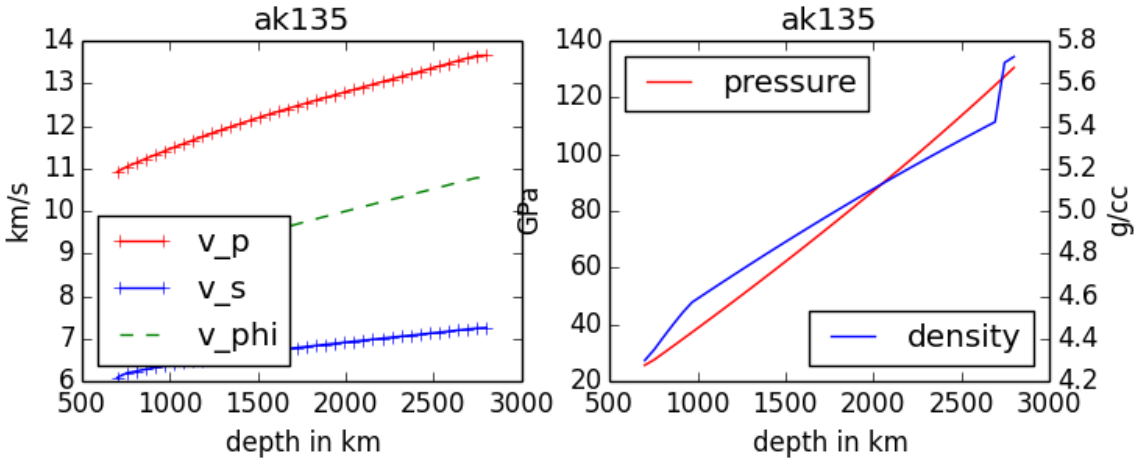
- *Seismic*

Demonstrates:

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

Resulting figures:





4.2.3 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 solid solutions
4. Defining your own solid 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

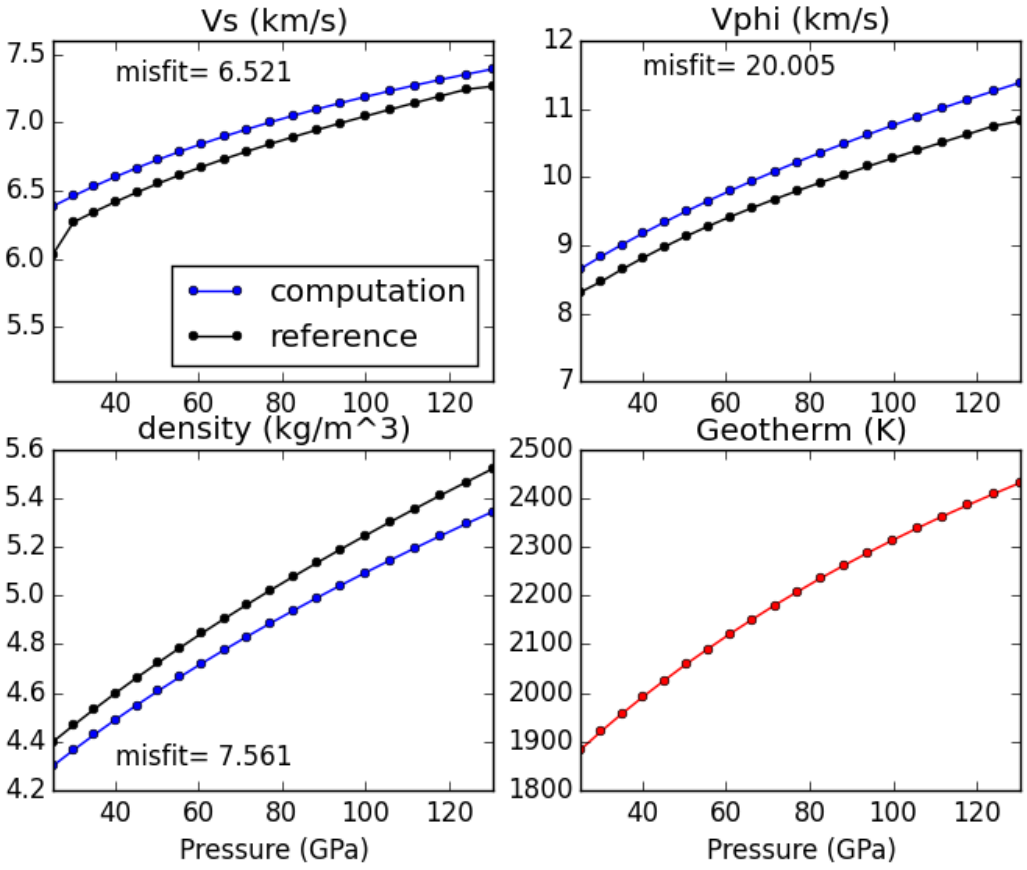
Uses:

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

Demonstrates:

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

Resulting figure:



4.2.4 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.

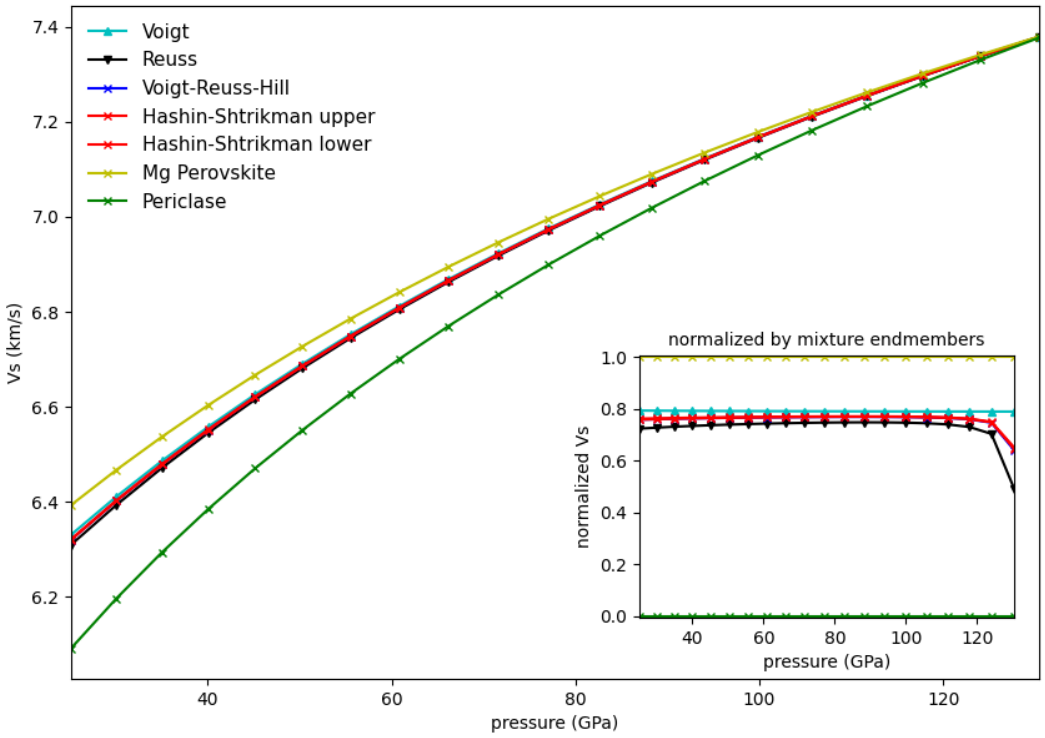
Specifically uses:

- *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:



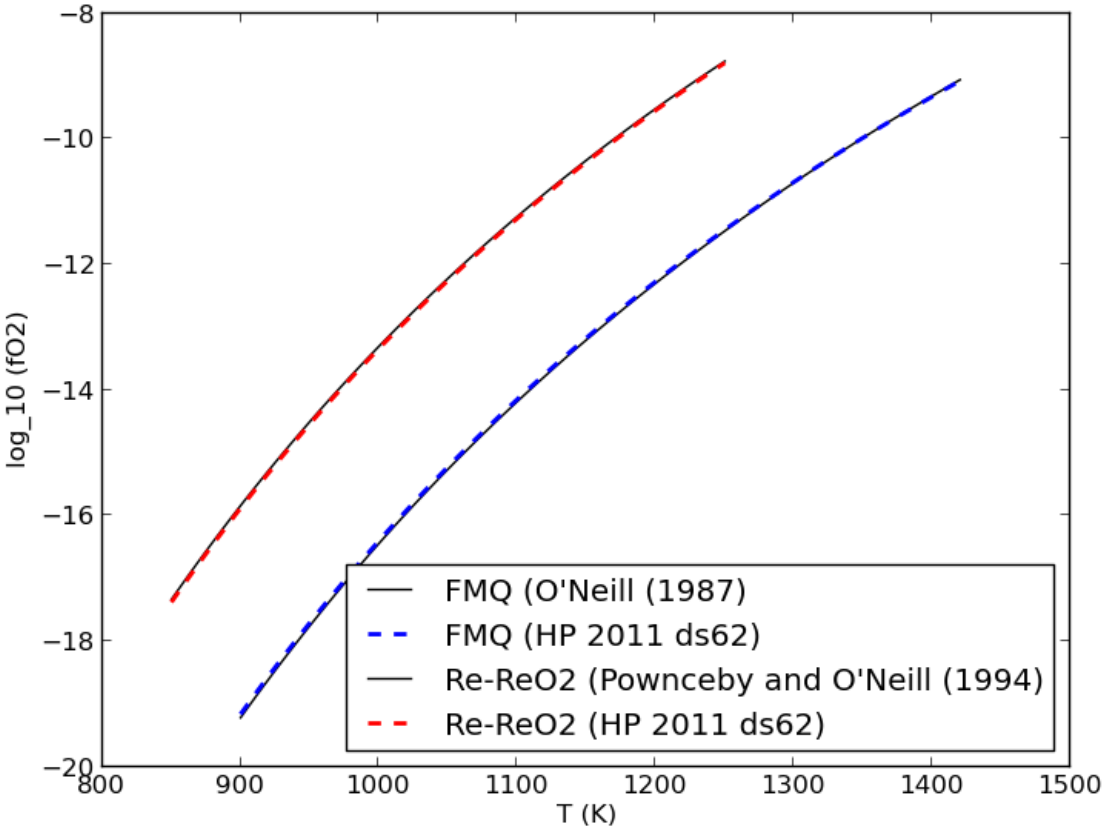
4.2.5 example_chemical_potentials

This example shows how to use the chemical potentials library of functions.

Demonstrates:

- How to calculate chemical potentials
- How to compute fugacities and relative fugacities

Resulting figure:



4.3 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_eos*,,
- *example_fit_composition*, and
- *example_equilibrate*.

4.3.1 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.

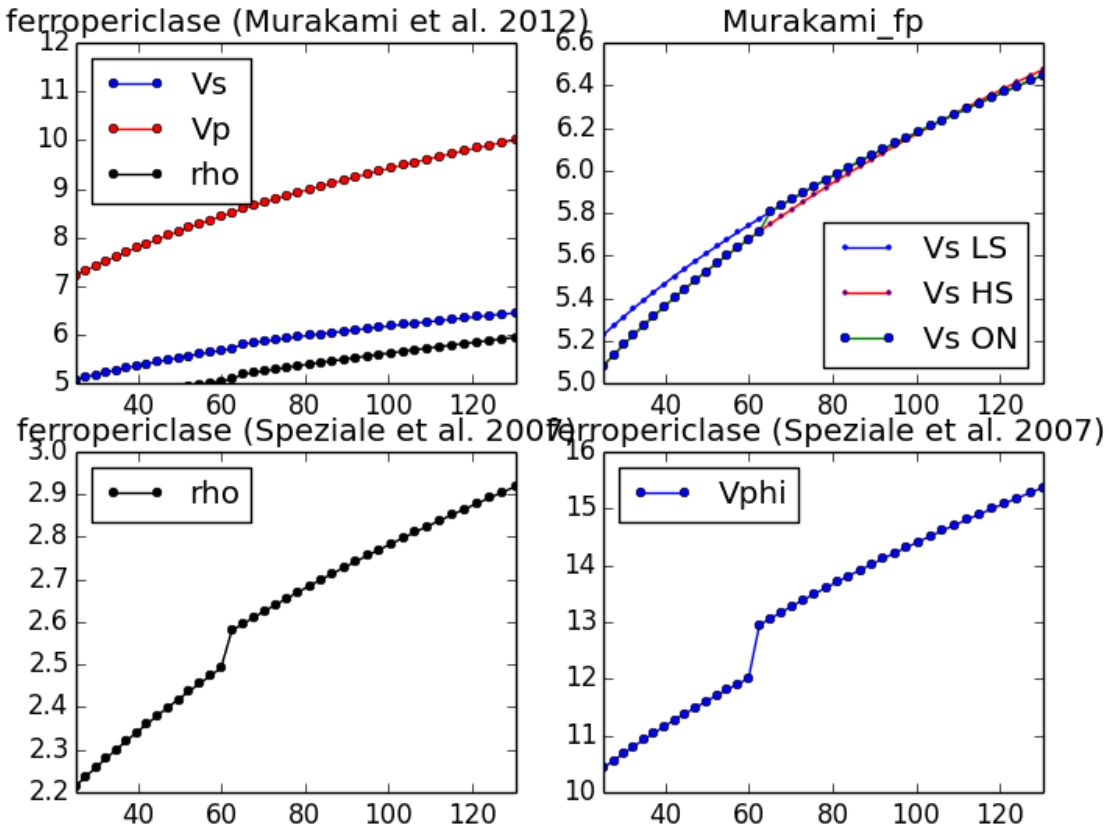
Specifically uses:

- `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:



4.3.2 example_spintransition_thermal

This example illustrates how to create a non-ideal solid 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 solid solution that incorporates all three endmembers. The modified solid 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.

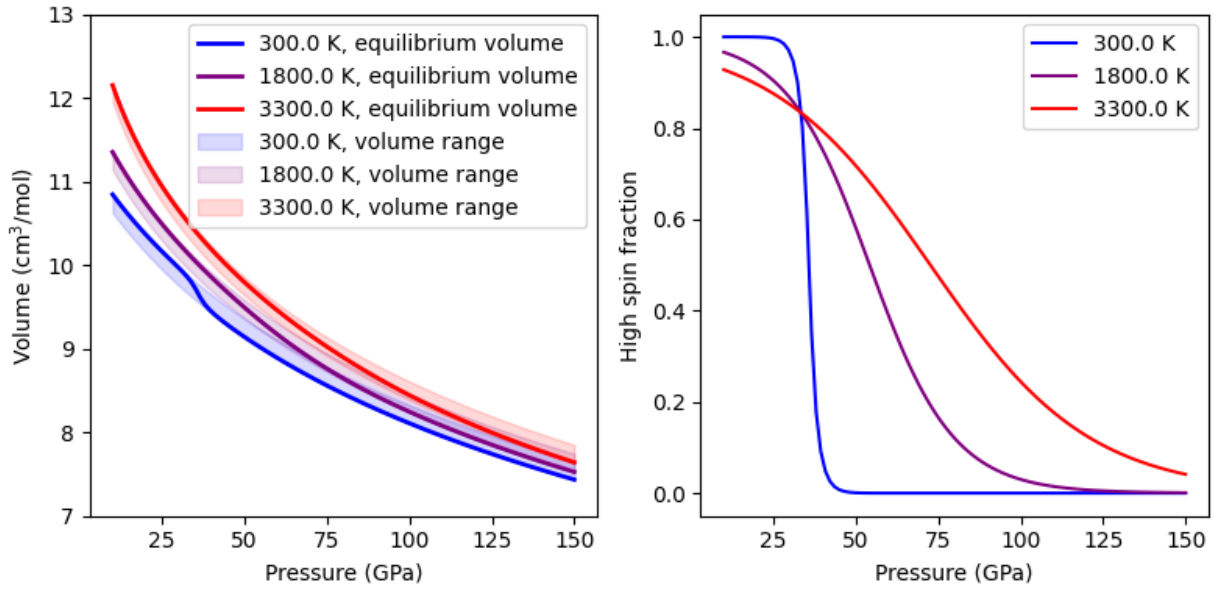
Specifically uses:

- *burnman.Mineral()*
- *burnman.SolidSolution()*

Demonstrates:

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

Resulting figure:



4.3.3 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

4.3.4 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

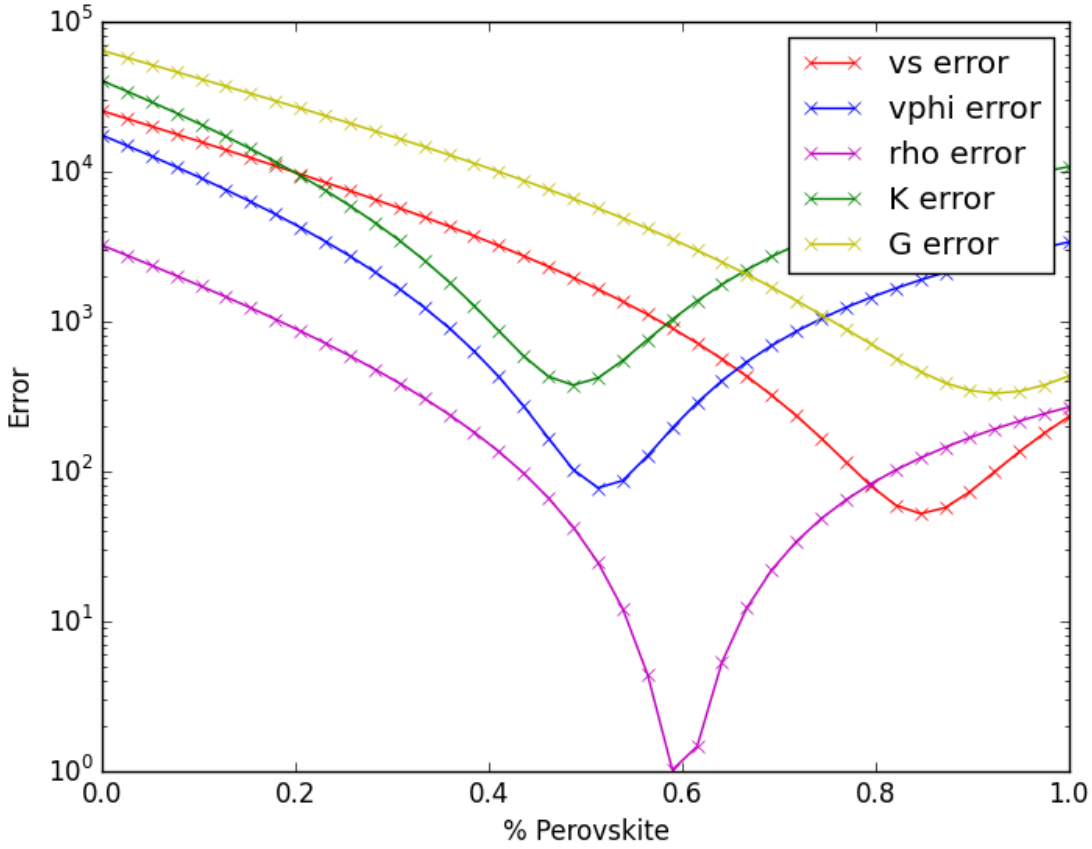
Uses:

- *Mineral databases*
- *burnman.Composite*
- `burnman.seismic.PREM`
- `burnman.geotherm.brown_shankland()`
- *burnman.Material.evaluate()*
- *burnman.tools.math.compare_l2()*

Demonstrates:

- compare errors between models
- loops over models

Resulting figure:



4.3.5 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.

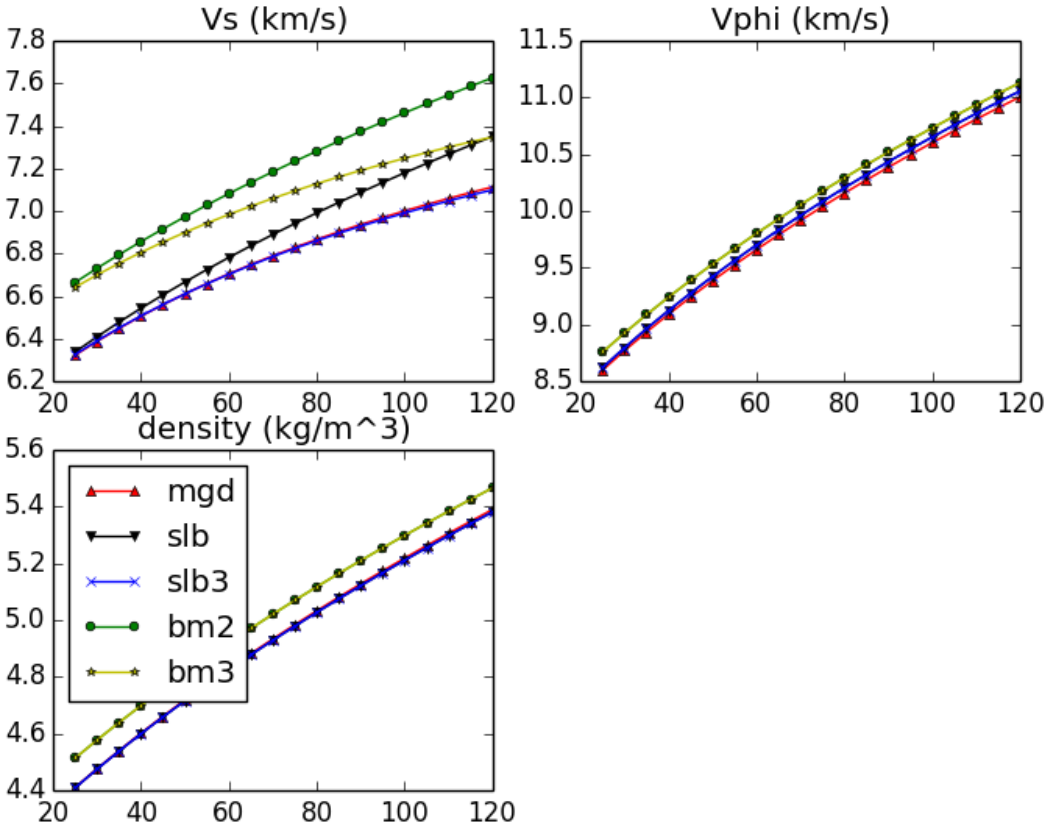
Specifically uses:

- *Equations of state*

Demonstrates:

- Each method for calculating velocity profiles currently included within BurnMan

Resulting figure:



4.3.6 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.

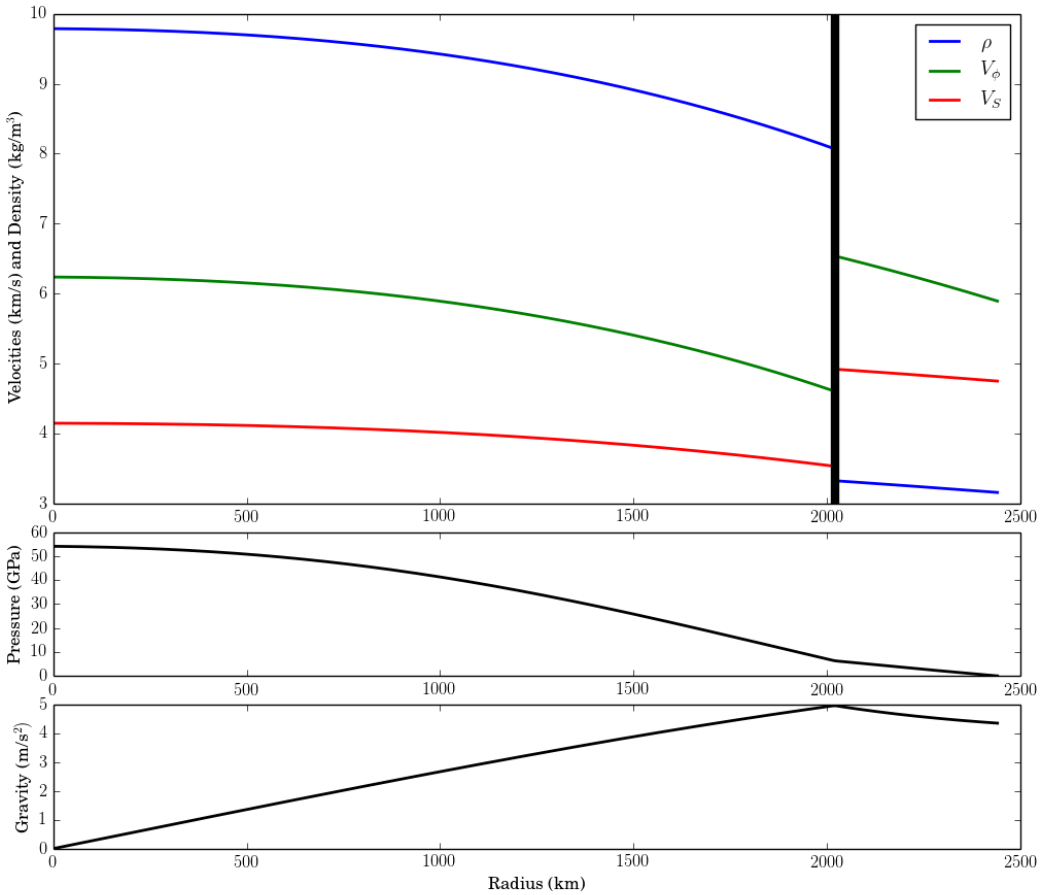
Uses:

- *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:



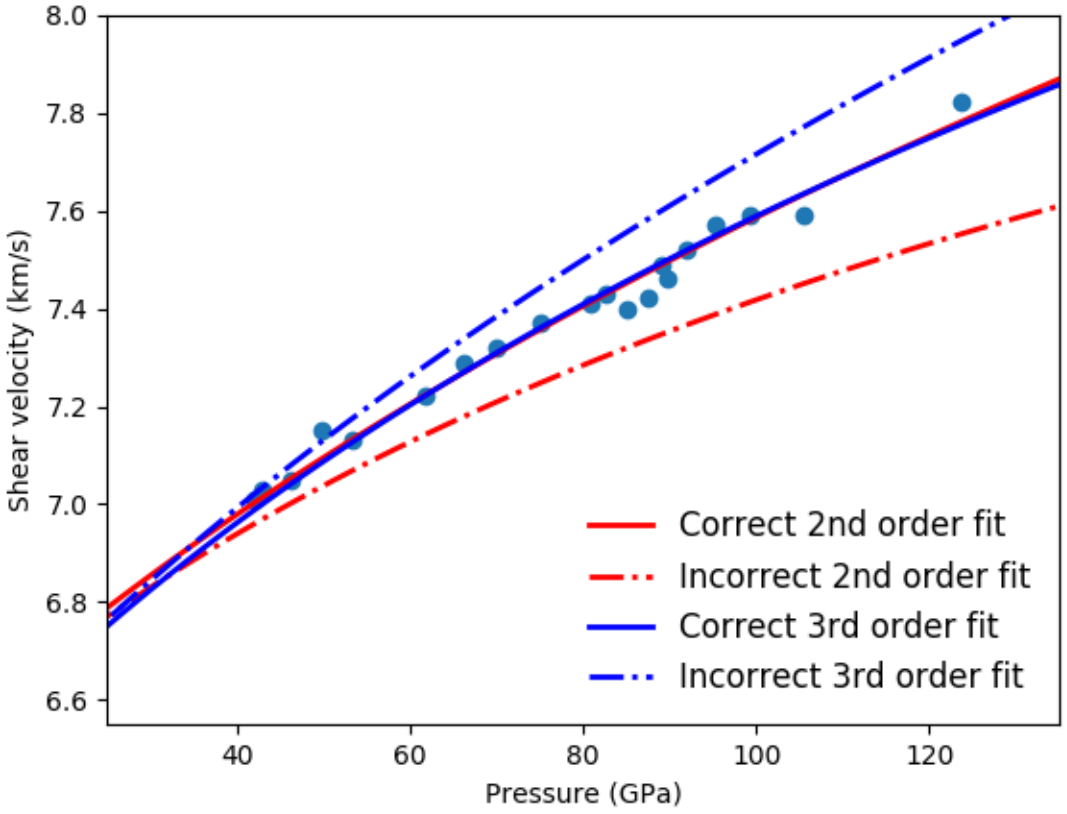
4.3.7 example_fit_data

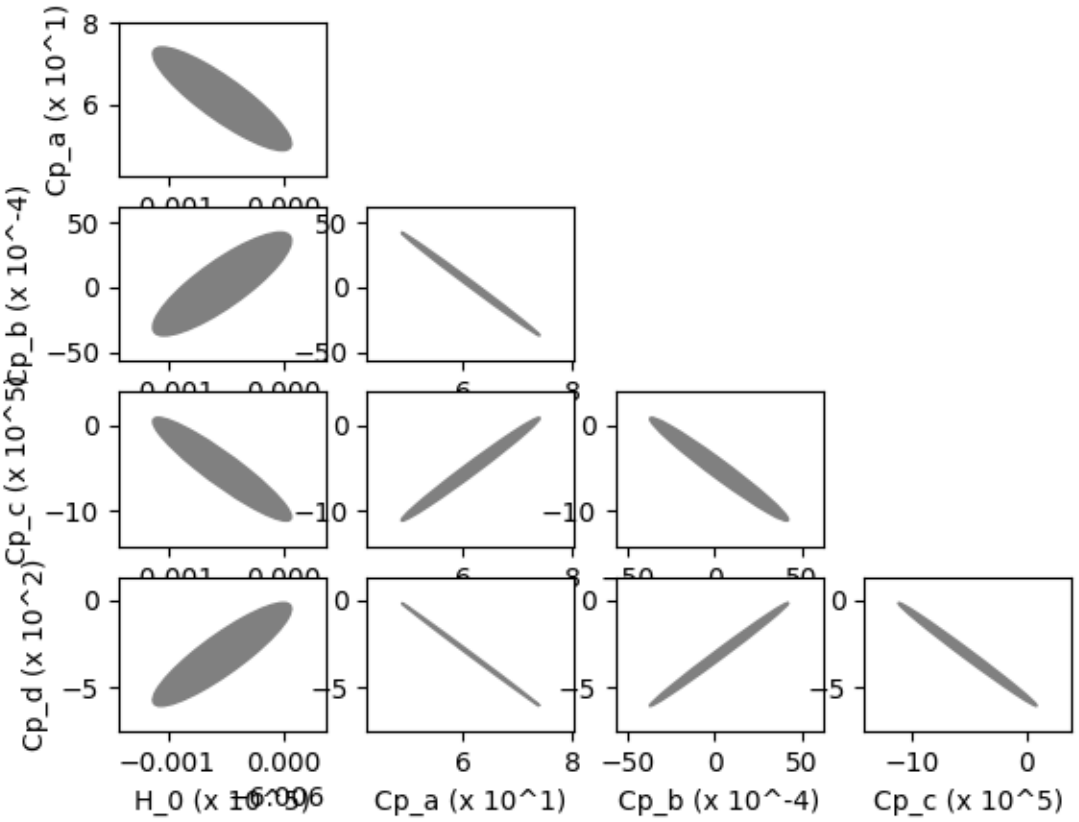
This example demonstrates BurnMan's functionality to fit various mineral physics data to an EoS of the user's choice.

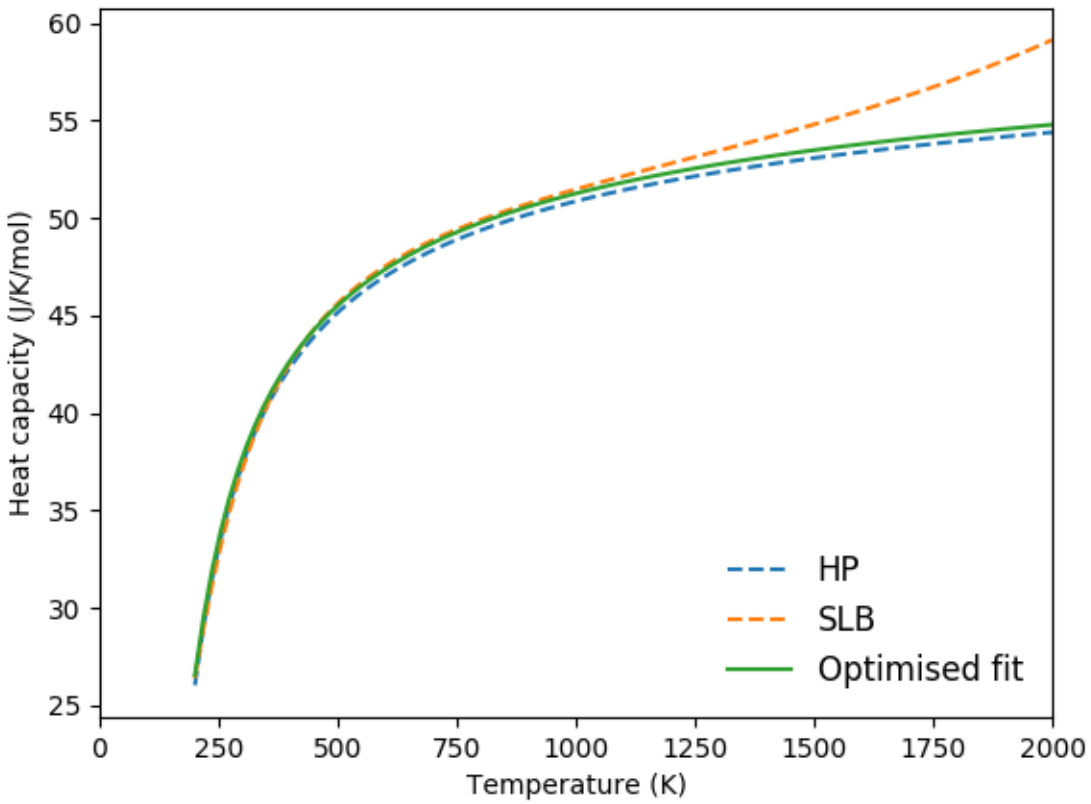
Please note also the separate file example_fit_eos.py, which can be viewed as a more advanced example in the same general field.

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 10^6)
Cp_a (x 10^1)
Cp_b (x 10^-4)
Cp_c (x 10^5)



4.3.8 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.

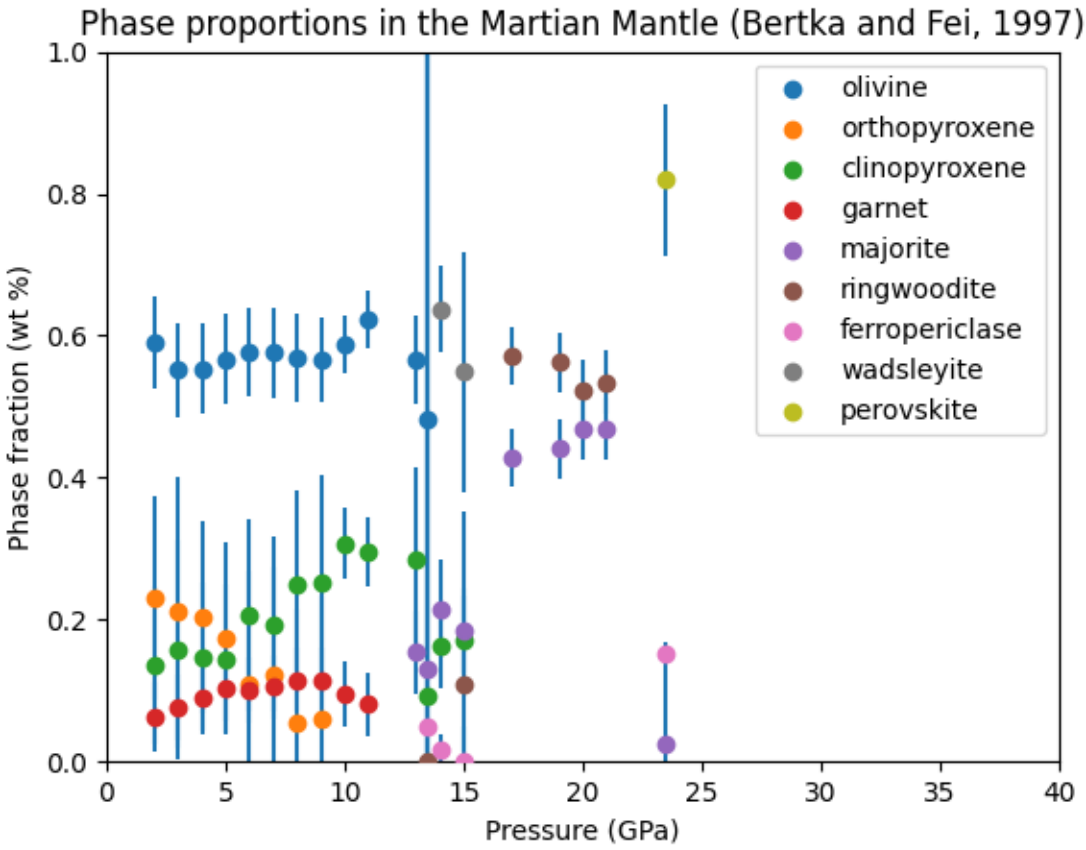
Uses:

- *burnman.Composition*
- *burnman.SolidSolution*
- *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:



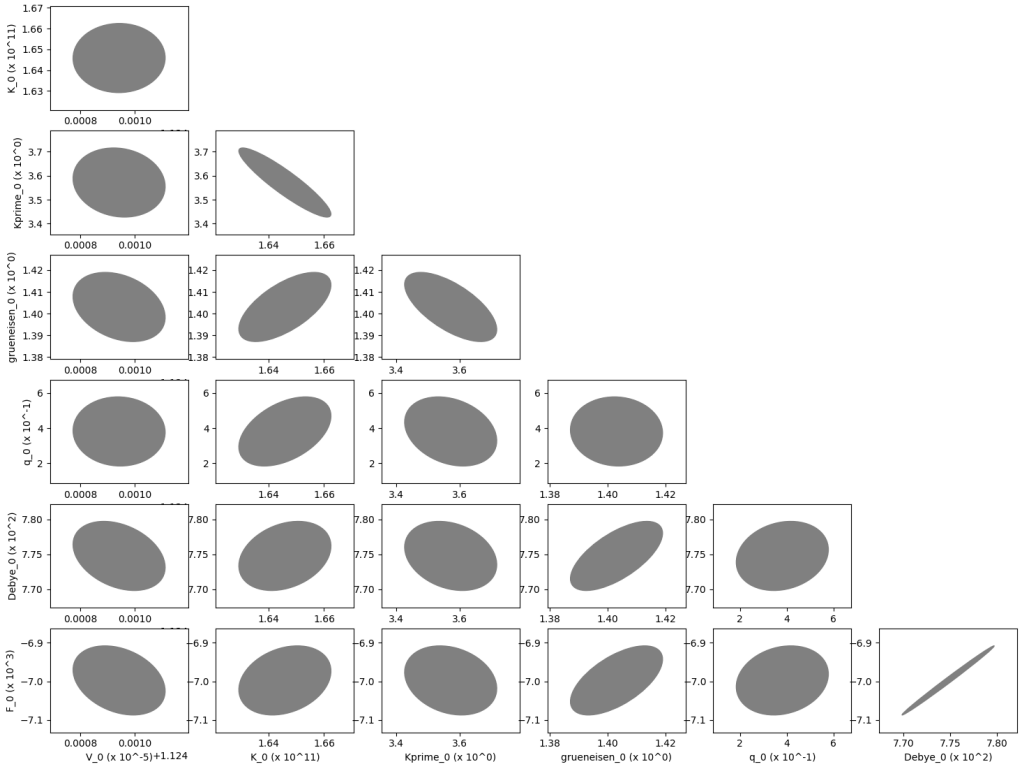
4.3.9 example_fit_eos

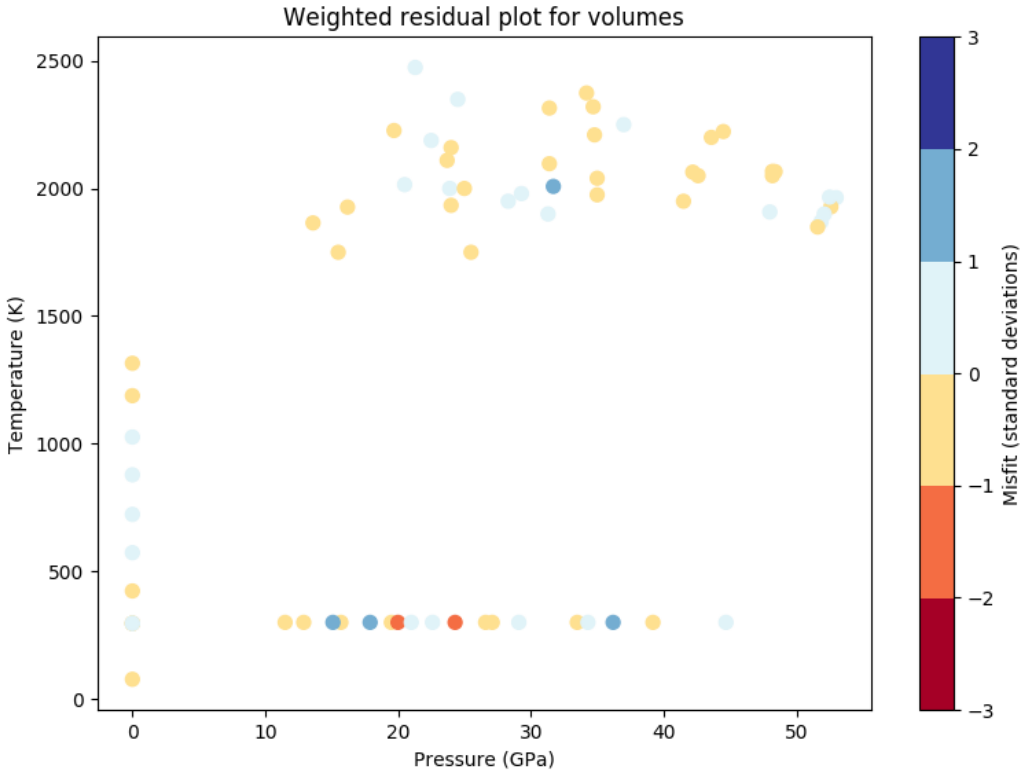
This example demonstrates BurnMan's functionality to fit data to an EoS of the user's choice.

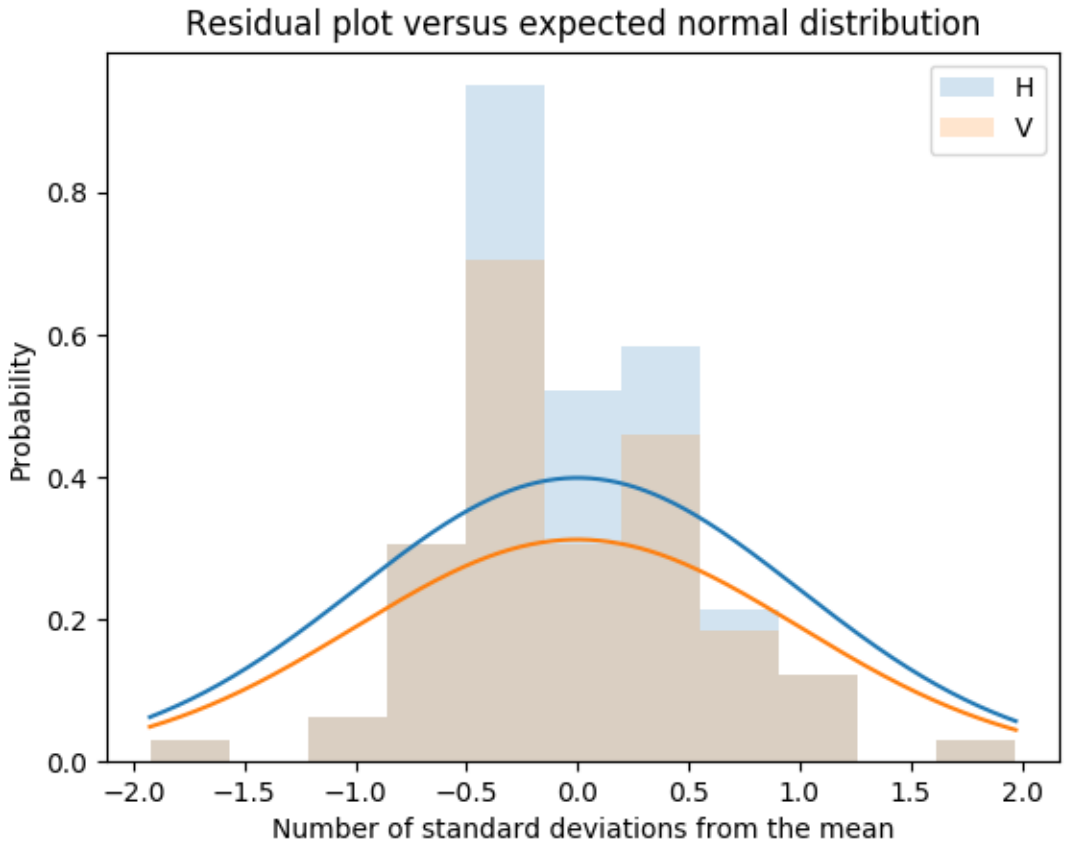
The first example deals with simple PVT fitting. The second example illustrates how powerful it can be to provide non-PVT constraints to the same fitting problem.

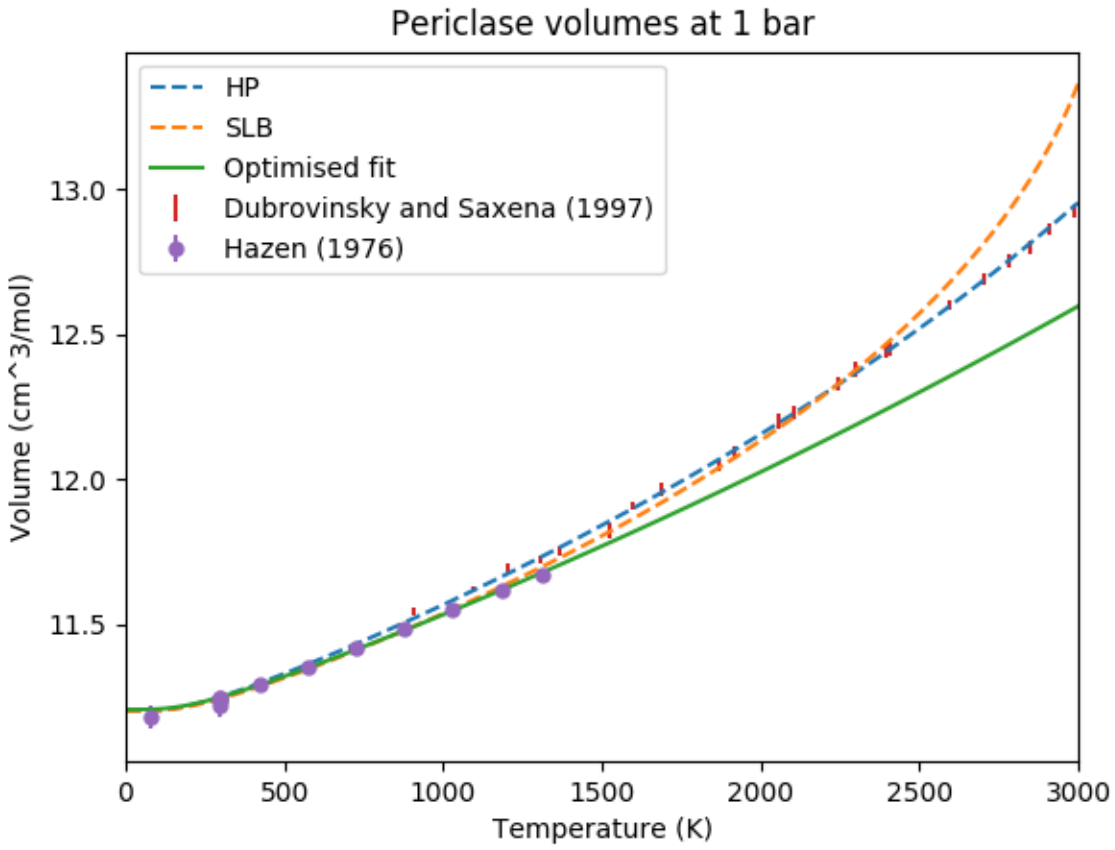
teaches: - least squares fitting

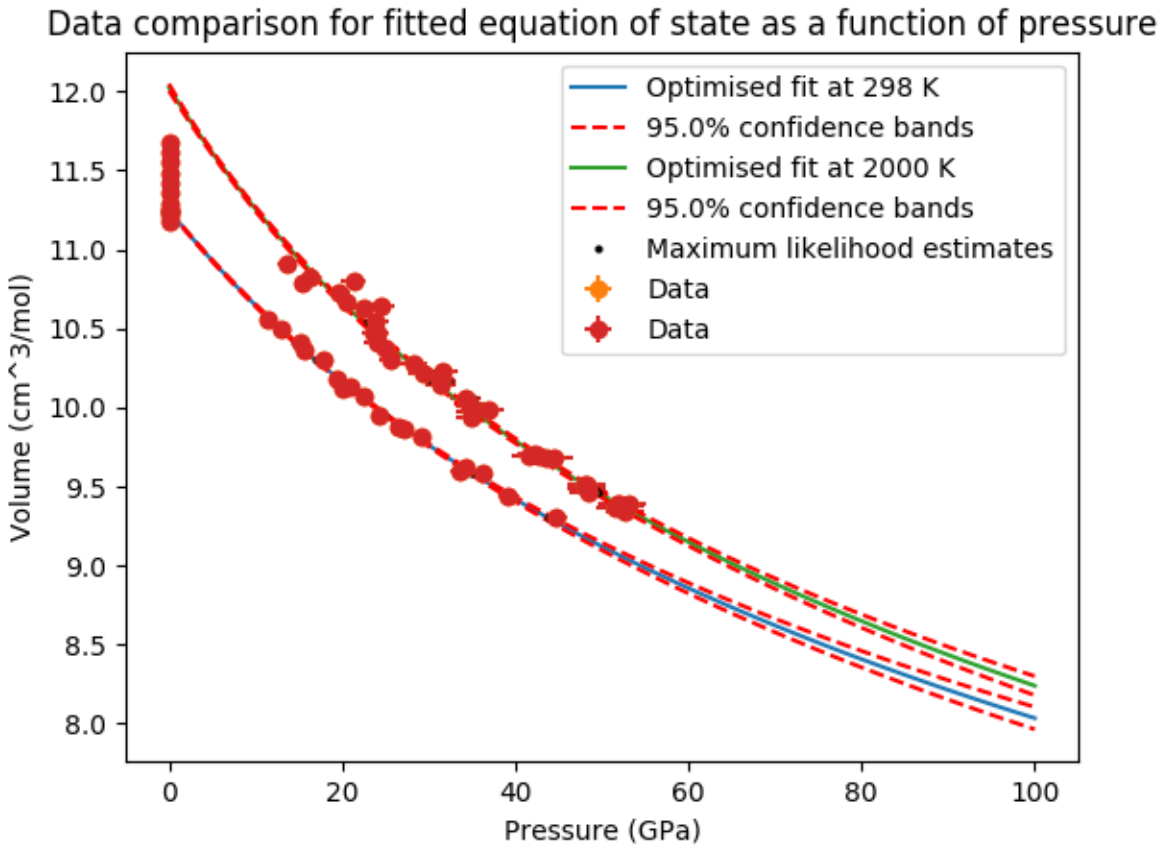
Last seven resulting figures:

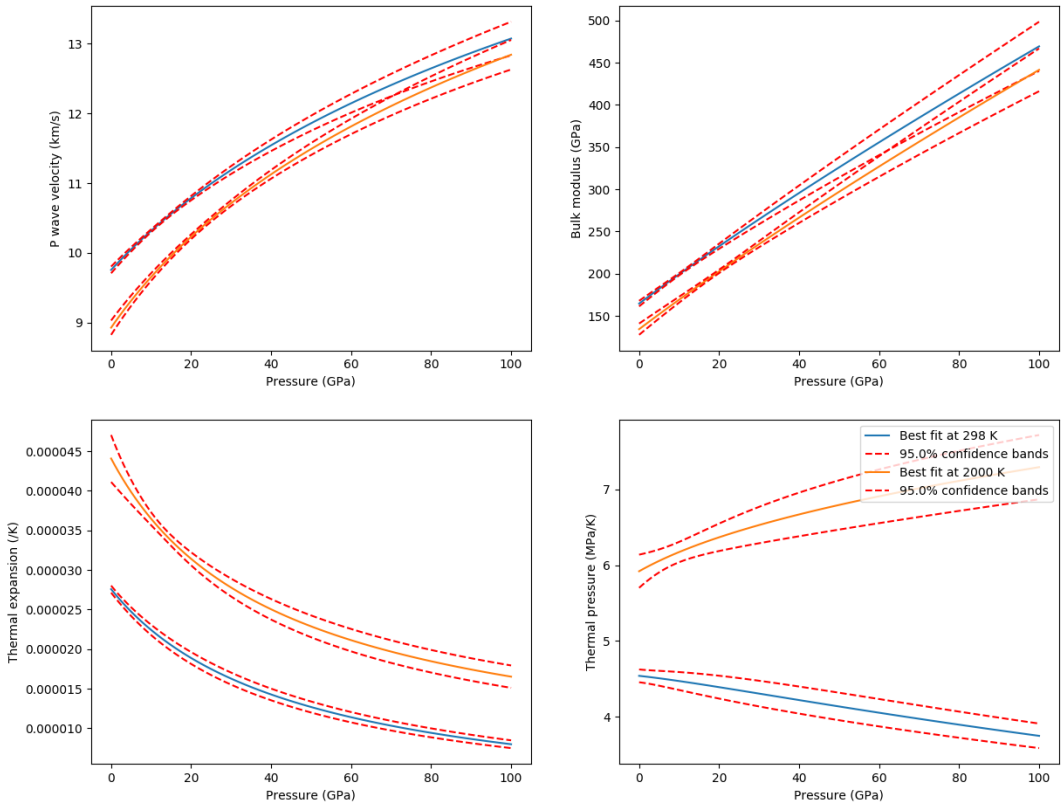

K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)
F_0 (x 10^3)
V_0 (x 10^-5)+1.124
K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)

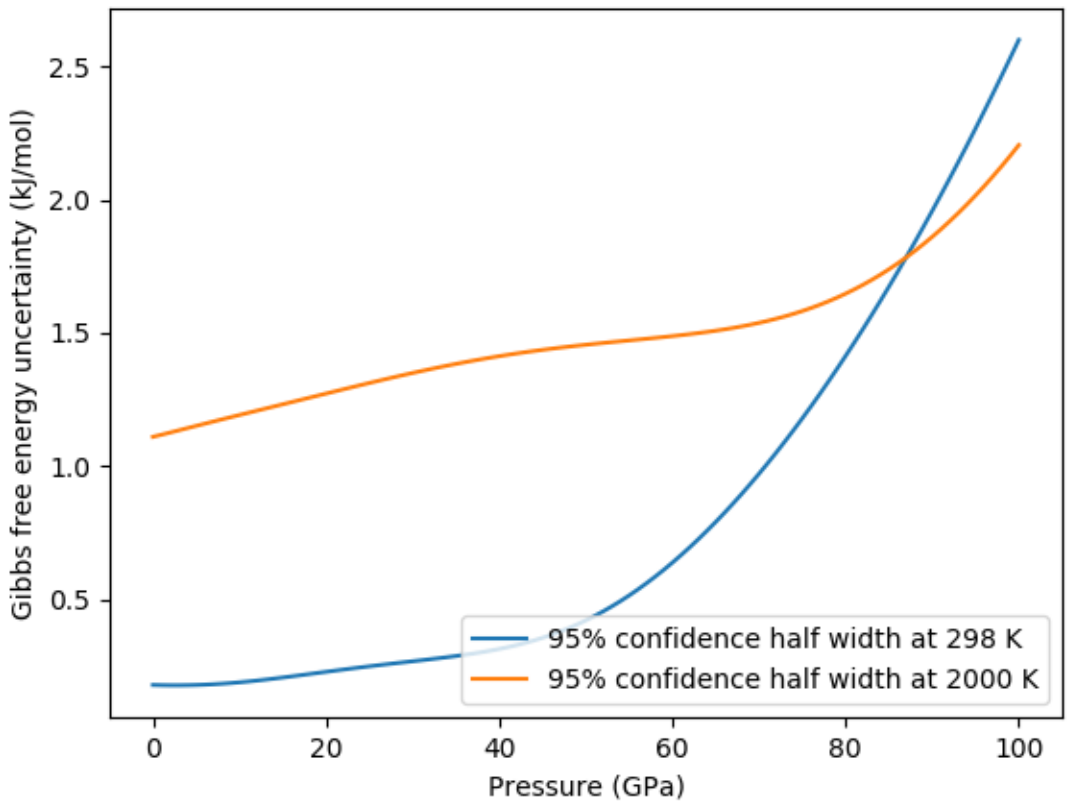

Weighted residual plot for volumes
Temperature (K)
Pressure (GPa)
Misfit (standard deviations)
2500
2000
1500
1000
500
0
0
10
20
30
40
50
3
2
1
0
−1
−2
−3


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



4.3.10 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.equilibrate()`

Resulting figures:

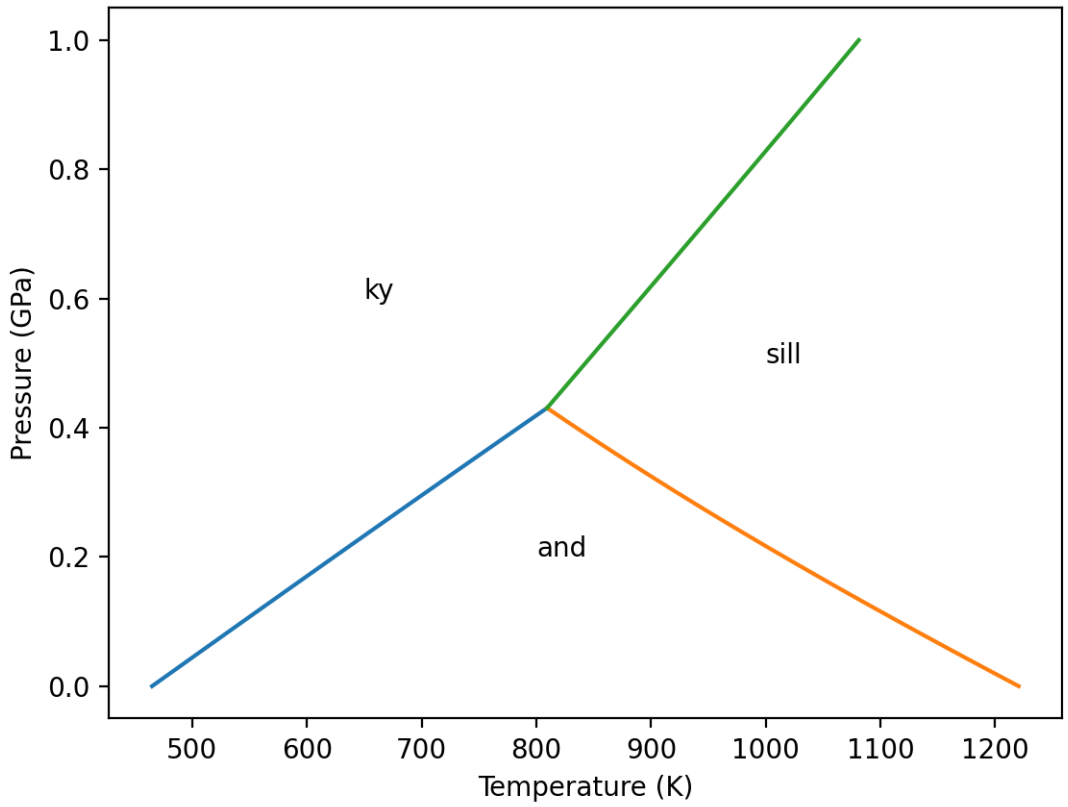


Fig. 1: The classic aluminosilicate diagram.

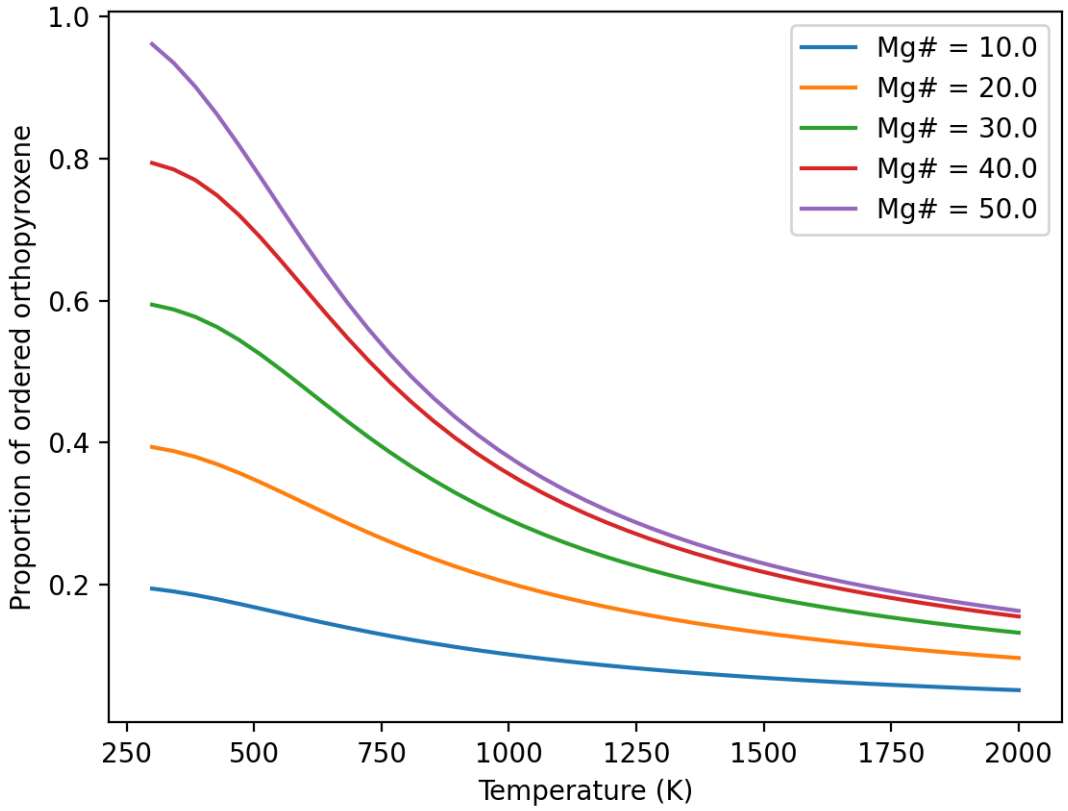


Fig. 2: Ordering in two site orthopyroxene.

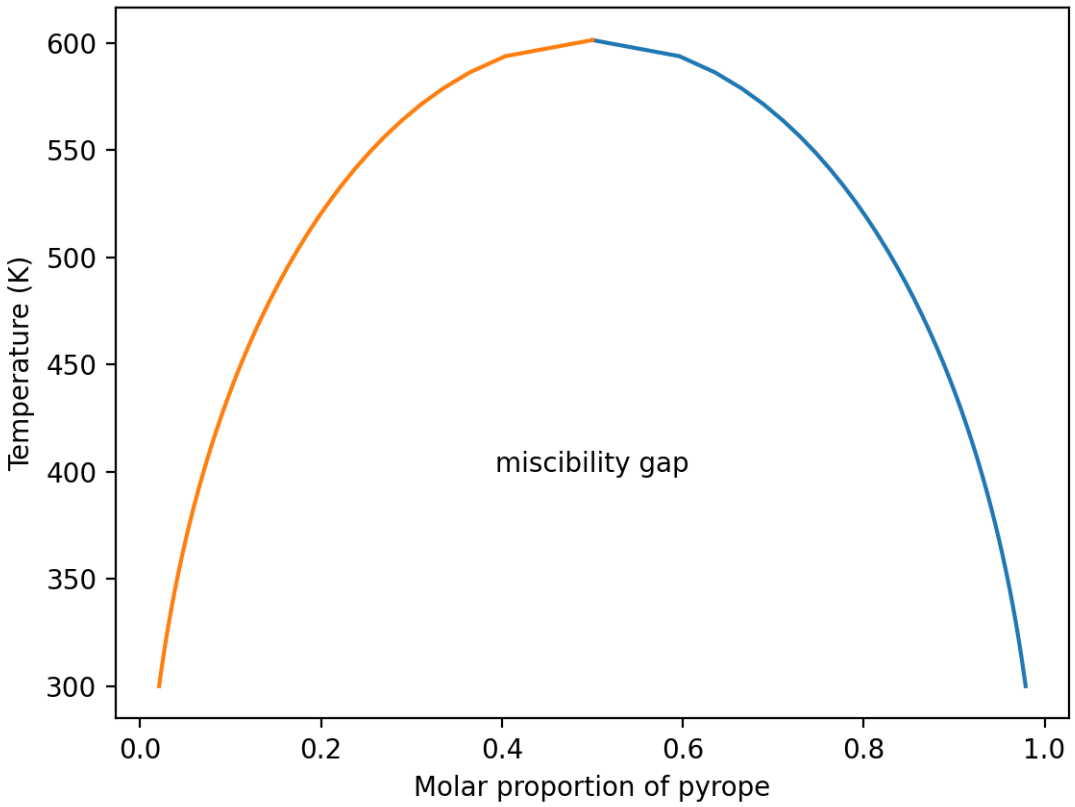


Fig. 3: Miscibility in the pyrope-grossular garnet system

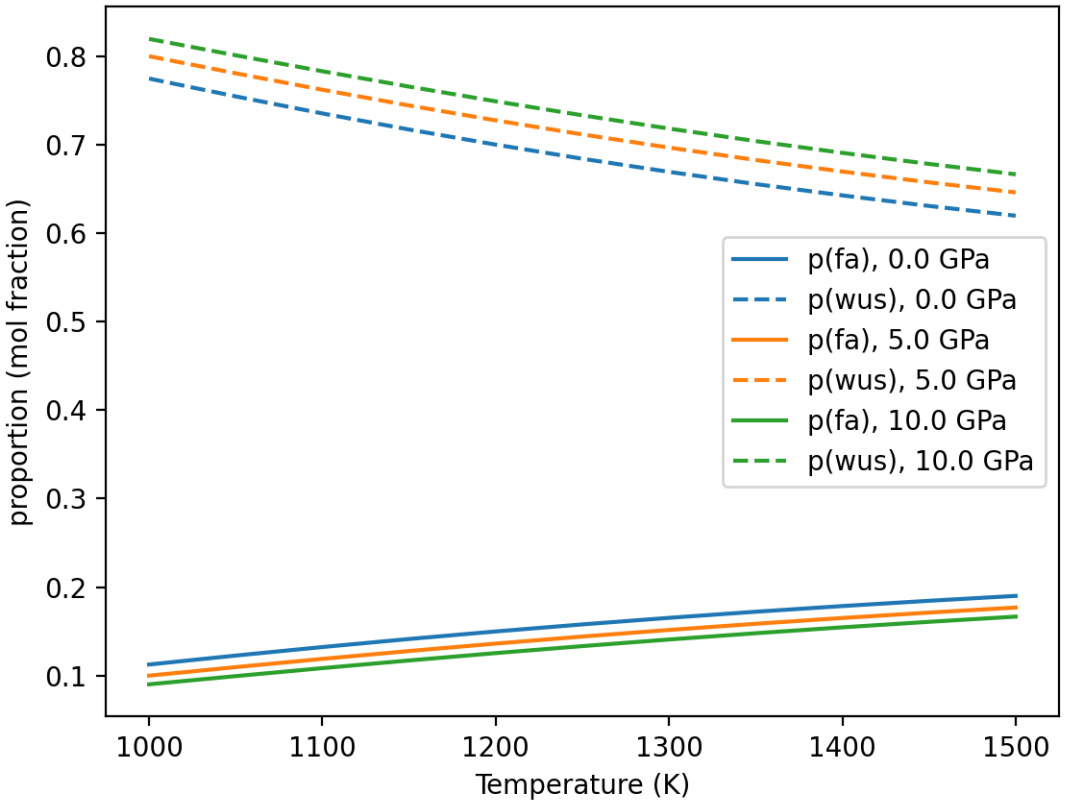


Fig. 4: Mg-Fe partitioning between olivine and ferropericlase.

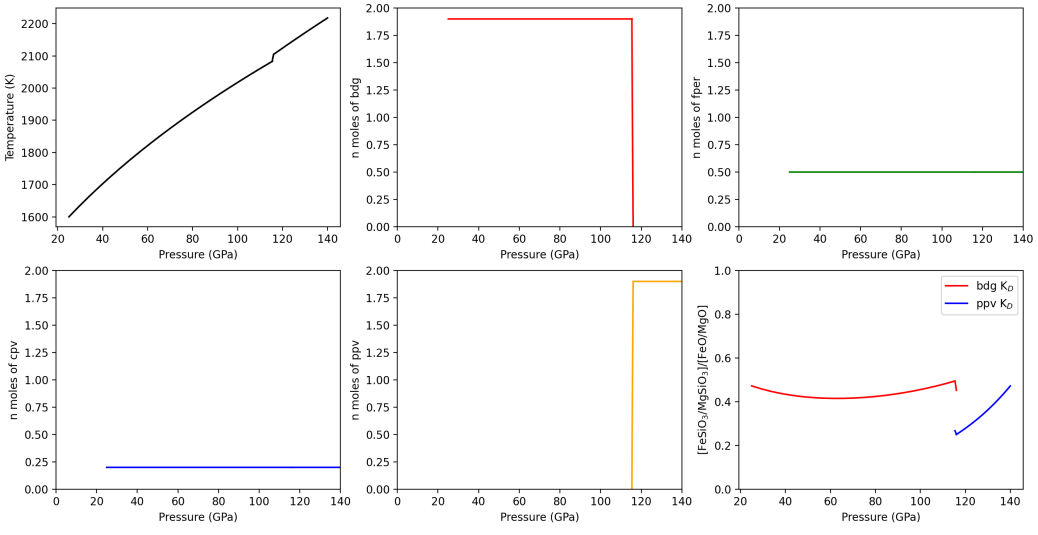


Fig. 5: Phase equilibria in the lower mantle.

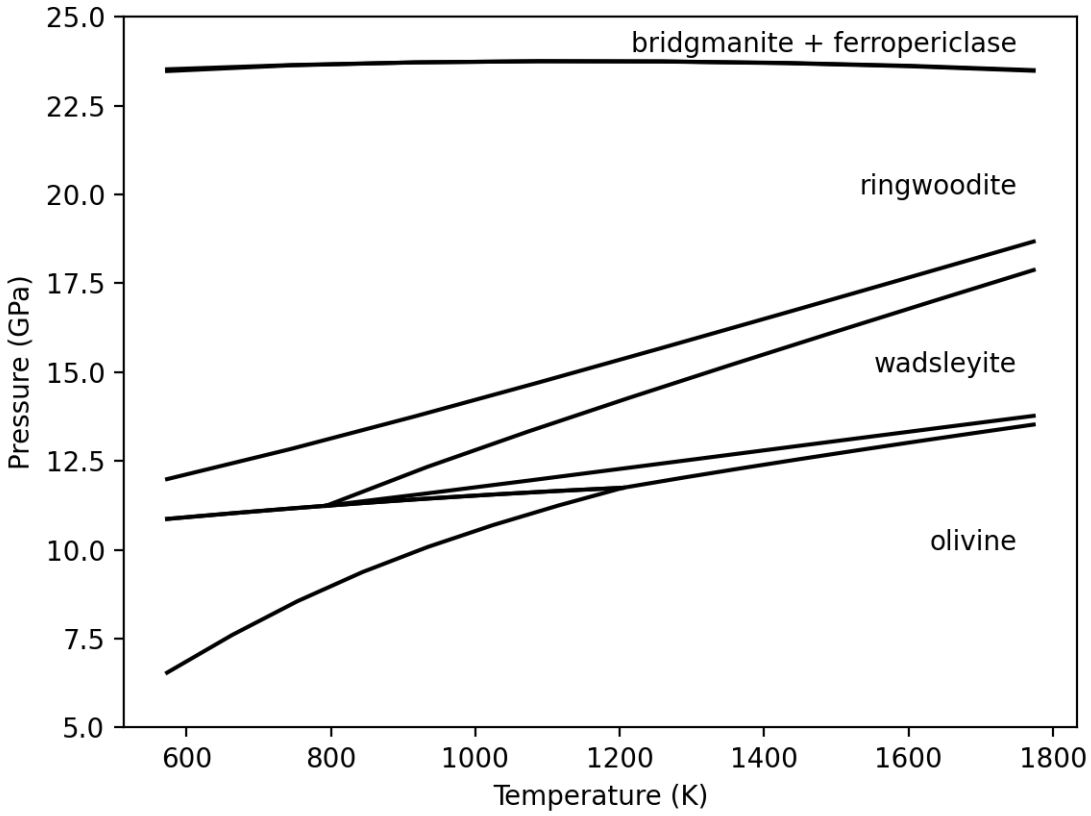


Fig. 6: A P-T pseudosection for a composition of Fe0.2Mg1.8SiO4 (fo90).

4.4 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]

4.4.1 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

4.4.2 paper_benchmark

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

4.4.3 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*)

4.4.4 paper_incorrect_averaging

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

4.4.5 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

4.4.6 paper_onefit

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

4.4.7 paper_uncertain

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

4.5 Misc or work in progress

4.5.1 example_grid

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

4.5.2 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

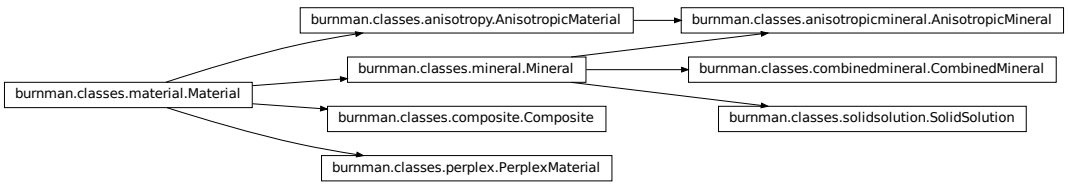
CHAPTER

FIVE

AUTOGENERATED FULL API

5.1 Materials

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



5.1.1 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.

Notes

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

name [string] Name of this material.

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].

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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

property molar_internal_energy
Returns the molar internal energy of the mineral.

Returns

molar_internal_energy [float] The internal energy in [J/mol].

Notes

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

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Returns

molar_gibbs [float] Gibbs free energy in [J/mol].

Notes

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

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Returns

molar_helmholtz [float] Helmholtz free energy in [J/mol].

Notes

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

property molar_mass

Returns molar mass of the mineral.

Returns

molar_mass [float] Molar mass in [kg/mol].

Notes

- Needs to be implemented in derived classes.

property molar_volume

Returns molar volume of the mineral.

Returns

molar_volume [float] Molar volume in [m^3/mol].

Notes

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

property density

Returns the density of this material.

Returns

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

Notes

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

property molar_entropy

Returns molar entropy of the mineral.

Returns

molar_entropy [float] Entropy in [J/K/mol].

Notes

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

property molar_enthalpy

Returns molar enthalpy of the mineral.

Returns

molar_enthalpy [float] Enthalpy in [J/mol].

Notes

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

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Returns

isothermal_bulk_modulus [float] Bulk modulus in [Pa].

Notes

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

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Returns

adiabatic_bulk_modulus [float] Adiabatic bulk modulus in [Pa].

Notes

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

property isothermal_compressibility

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

Returns

(K_T)^-1 [float] Compressibility in [1/Pa].

Notes

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

property adiabatic_compressibility

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

Returns

adiabatic_compressibility [float] adiabatic compressibility in [1/Pa].

Notes

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

property shear_modulus

Returns shear modulus of the mineral.

Returns

shear_modulus [float] Shear modulus in [Pa].

Notes

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

property p_wave_velocity

Returns P wave speed of the mineral.

Returns

p_wave_velocity [float] P wave speed in [m/s].

Notes

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

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Returns

bulk sound velocity: float Sound velocity in [m/s].

Notes

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

property shear_wave_velocity

Returns shear wave speed of the mineral.

Returns

shear_wave_velocity [float] Wave speed in [m/s].

Notes

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

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Returns

gr [float] Grueneisen parameters [unitless].

Notes

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

property thermal_expansivity
Returns thermal expansion coefficient of the mineral.

Returns

alpha [float] Thermal expansivity in [1/K].

Notes

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

property molar_heat_capacity_v
Returns molar heat capacity at constant volume of the mineral.

Returns

molar_heat_capacity_v [float] Heat capacity in [J/K/mol].

Notes

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

property molar_heat_capacity_p
Returns molar heat capacity at constant pressure of the mineral.

Returns

molar_heat_capacity_p [float] Heat capacity in [J/K/mol].

Notes

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

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()*

5.1.2 Perple_X Class

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

Bases: *burnman.classes.material.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.

Returns

molar_volume [float] Molar volume in [m^3/mol].

Notes

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

property molar_enthalpy

Returns molar enthalpy of the mineral.

Returns

molar_enthalpy [float] Enthalpy in [J/mol].

Notes

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

property molar_entropy

Returns molar entropy of the mineral.

Returns

molar_entropy [float] Entropy in [J/K/mol].

Notes

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

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Returns

isothermal_bulk_modulus [float] Bulk modulus in [Pa].

Notes

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

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Returns

adiabatic_bulk_modulus [float] Adiabatic bulk modulus in [Pa].

Notes

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

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Returns

molar_heat_capacity_p [float] Heat capacity in [J/K/mol].

Notes

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

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Returns

alpha [float] Thermal expansivity in [1/K].

Notes

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

property shear_modulus

Returns shear modulus of the mineral.

Returns

shear_modulus [float] Shear modulus in [Pa].

Notes

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

property p_wave_velocity

Returns P wave speed of the mineral.

Returns

p_wave_velocity [float] P wave speed in [m/s].

Notes

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

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Returns

bulk sound velocity: float Sound velocity in [m/s].

Notes

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

property shear_wave_velocity

Returns shear wave speed of the mineral.

Returns

shear_wave_velocity [float] Wave speed in [m/s].

Notes

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

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Returns

molar_gibbs [float] Gibbs free energy in [J/mol].

Notes

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

property molar_mass

Returns molar mass of the mineral.

Returns

molar_mass [float] Molar mass in [kg/mol].

Notes

- Needs to be implemented in derived classes.

property density

Returns the density of this material.

Returns

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

Notes

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

property molar_internal_energy

Returns the molar internal energy of the mineral.

Returns

molar_internal_energy [float] The internal energy in [J/mol].

Notes

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

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Returns

molar_helmholtz [float] Helmholtz free energy in [J/mol].

Notes

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

property isothermal_compressibility

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

Returns

(K_T)^-1 [float] Compressibility in [1/Pa].

Notes

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

property adiabatic_compressibility

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

Returns

adiabatic_compressibility [float] adiabatic compressibility in [1/Pa].

Notes

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

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Returns

molar_heat_capacity_v [float] Heat capacity in [J/K/mol].

Notes

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

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

Returns

gr [float] Grueneisen parameters [unitless].

Notes

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

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 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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property helmholtz
Alias for *molar_helmholtz()*

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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.

Notes

Needs to be implemented in derived classes.

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

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

name [string] Name 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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

5.1.3 Minerals

5.1.3.1 Endmembers

class burnman.**Mineral**(*params=None*, *property_modifiers=None*)
Bases: *burnman.classes.material.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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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.

Returns

molar_gibbs [float] Gibbs free energy in [J/mol].

Notes

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

property molar_volume

Returns molar volume of the mineral.

Returns

molar_volume [float] Molar volume in [m^3/mol].

Notes

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

property molar_entropy

Returns molar entropy of the mineral.

Returns

molar_entropy [float] Entropy in [J/K/mol].

Notes

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

property isothermal_bulk_modulus

Returns isothermal bulk modulus of the material.

Returns

isothermal_bulk_modulus [float] Bulk modulus in [Pa].

Notes

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

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Returns

molar_heat_capacity_p [float] Heat capacity in [J/K/mol].

Notes

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

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Returns

alpha [float] Thermal expansivity in [1/K].

Notes

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

property shear_modulus

Returns shear modulus of the mineral.

Returns

shear_modulus [float] Shear modulus in [Pa].

Notes

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

property formula

Returns the chemical formula of the Mineral class

property molar_mass

Returns molar mass of the mineral.

Returns

molar_mass [float] Molar mass in [kg/mol].

Notes

- Needs to be implemented in derived classes.

property density

Returns the density of this material.

Returns

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

Notes

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

property molar_internal_energy

Returns the molar internal energy of the mineral.

Returns

molar_internal_energy [float] The internal energy in [J/mol].

Notes

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

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Returns

molar_helmholtz [float] Helmholtz free energy in [J/mol].

Notes

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

property molar_enthalpy

Returns molar enthalpy of the mineral.

Returns

molar_enthalpy [float] Enthalpy in [J/mol].

Notes

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

property adiabatic_bulk_modulus

Returns the adiabatic bulk modulus of the mineral.

Returns

adiabatic_bulk_modulus [float] Adiabatic bulk modulus in [Pa].

Notes

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

property isothermal_compressibility

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

Returns

(K_T)^-1 [float] Compressibility in [1/Pa].

Notes

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

property adiabatic_compressibility

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

Returns

adiabatic_compressibility [float] adiabatic compressibility in [1/Pa].

Notes

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

property p_wave_velocity

Returns P wave speed of the mineral.

Returns

p_wave_velocity [float] P wave speed in [m/s].

Notes

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

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

Returns

bulk sound velocity: float Sound velocity in [m/s].

Notes

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

property shear_wave_velocity
Returns shear wave speed of the mineral.

Returns

shear_wave_velocity [float] Wave speed in [m/s].

Notes

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

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 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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property grueneisen_parameter
Returns the grueneisen parameter of the mineral.

Returns

gr [float] Grueneisen parameters [unitless].

Notes

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

property helmholtz
Alias for *molar_helmholtz()*

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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()*

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

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.

Returns

molar_heat_capacity_v [float] Heat capacity in [J/K/mol].

Notes

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

5.1.3.2 Solid solutions

class burnman.**SolidSolution**(*name=None*, *solution_type=None*, *endmembers=None*, *energy_interaction=None*, *volume_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *volume_ternary_terms=None*, *entropy_ternary_terms=None*, *alphas=None*, *molar_fractions=None*)

Bases: *burnman.classes.mineral.Mineral*

This is the base class for all solid 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 solid solution can only be queried after setting the pressure, temperature and composition using set_state().

This class is available as *burnman.SolidSolution*. It uses an instance of *burnman.SolutionModel* to calculate interaction terms between endmembers.

All the solid 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 solid solution

solution_type [string] String determining which SolutionModel to use. One of 'mechanical', 'ideal', 'symmetric', 'asymmetric' or 'subregular'.

endmembers [list of lists] List of endmembers in this solid solution. The first item of each list should be a *burnman.Mineral* object. The second item should be a string with the site formula of the endmember.

molar_fractions [numpy array (optional)] The molar fractions of each endmember in the solid 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.

get_endmembers()

set_composition(*molar_fractions*)

Set the composition for this solid 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 solid 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 solid solutions.

`property excess_partial_volumes`
Returns excess partial volumes [m^3]. Property specific to solid solutions.

`property excess_partial_entropies`
Returns excess partial entropies [J/K]. Property specific to solid solutions.

`property partial_gibbs`
Returns excess partial molar gibbs free energy [J/mol]. Property specific to solid solutions.

`property partial_volumes`
Returns excess partial volumes [m^3]. Property specific to solid solutions.

`property partial_entropies`
Returns excess partial entropies [J/K]. Property specific to solid solutions.

`property excess_gibbs`
Returns molar excess gibbs free energy [J/mol]. Property specific to solid solutions.

property gibbs_hessian
Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solid solutions.

property entropy_hessian
Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solid solutions.

property volume_hessian
Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solid solutions.

property molar_gibbs
Returns molar Gibbs free energy of the solid solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz
Returns molar Helmholtz free energy of the solid solution [J/mol]. Aliased with self.helmholtz.

property molar_mass
Returns molar mass of the solid solution [kg/mol].

property excess_volume
Returns excess molar volume of the solid solution [m^3/mol]. Specific property for solid solutions.

property molar_volume
Returns molar volume of the solid solution [m^3/mol]. Aliased with self.V.

property density
Returns density of the solid solution [kg/m^3]. Aliased with self.rho.

property excess_entropy
Returns excess molar entropy [J/K/mol]. Property specific to solid solutions.

property molar_entropy
Returns molar entropy of the solid solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy
Returns excess molar enthalpy [J/mol]. Property specific to solid solutions.

property molar_enthalpy
Returns molar enthalpy of the solid solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus
Returns isothermal bulk modulus of the solid solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus
Returns adiabatic bulk modulus of the solid solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility
Returns isothermal compressibility of the solid solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility
Returns adiabatic compressibility of the solid solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus
Returns shear modulus of the solid solution [Pa]. Aliased with self.G.

property p_wave_velocity
Returns P wave speed of the solid solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity
Returns bulk sound speed of the solid solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity
Returns shear wave speed of the solid solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter
Returns grueneisen parameter of the solid solution [unitless]. Aliased with self.gr.

property thermal_expansivity
Returns thermal expansion coefficient (alpha) of the solid solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v
Returns molar heat capacity at constant volume of the solid solution [J/K/mol]. Aliased with self.C_v.

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 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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property helmholtz
Alias for *molar_helmholtz()*

property molar_heat_capacity_p
Returns molar heat capacity at constant pressure of the solid solution [J/K/mol]. Aliased with self.C_p.

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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()*

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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 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 solid solution.

property endmember_formulae
A list of formulae for all the endmember in the solid solution.

property endmember_names
A list of names for all the endmember in the solid solution.

property n_endmembers
The number of endmembers in the solid solution.

property elements
A list of the elements which could be contained in the solid solution, returned in the IUPAC element order.

5.1.3.3 Mineral helpers

class burnman.classes.mineral_helpers.**HelperSpinTransition**(*transition_pressure*, *ls_mat*, *hs_mat*)

Bases: *burnman.classes.composite.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_compressibility
Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_S

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

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 SolidSolution.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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

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_compressibility
Returns isothermal compressibility of the composite (or inverse isothermal bulk modulus) [1/Pa] Aliased with self.beta_T

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()*.

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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_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

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()*.

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

5.1.3.4 Anisotropic materials

class burnman.**AnisotropicMaterial**(*rho*, *cijs*)
Bases: *burnman.classes.material.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.

Returns

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

Notes

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

property isentropic_bulk_modulus_voigt
Computes the isentropic bulk modulus (Voigt bound)

property isentropic_bulk_modulus_reuss
Computes the isentropic bulk modulus (Reuss bound)

property isentropic_bulk_modulus_vrh
Computes the isentropic bulk modulus (Voigt-Reuss-Hill average)

property isentropic_shear_modulus_voigt
Computes the isentropic shear modulus (Voigt bound)

property isentropic_shear_modulus_reuss
Computes the isentropic shear modulus (Reuss bound)

property isentropic_shear_modulus_vrh
Computes the shear modulus (Voigt-Reuss-Hill average)

property isentropic_universal_elastic_anisotropy
Compute the universal elastic anisotropy

property isentropic_isotropic_poisson_ratio
Compute mu, the isotropic Poisson ratio (a description of the laterial response to loading)

christoffel_tensor(*propagation_direction*)
Computes 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

isentropic_linear_compressibility(*direction*)
Computes the linear isentropic compressibility in a given direction relative to the stiffness tensor

isentropic_youngs_modulus(*direction*)
Computes the isentropic Youngs modulus in a given direction relative to the stiffness tensor

isentropic_shear_modulus(*plane_normal*, *shear_direction*)
Computes the isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)
Computes the isentropic poisson ratio given loading and response directions relative to the stiffness tensor

wave_velocities(*propagation_direction*)
Computes the compressional wave velocity, and two shear wave velocities in a given propagation direction

Returns two lists, 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.

Returns

adiabatic_bulk_modulus [float] Adiabatic bulk modulus in [Pa].

Notes

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

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

Returns

adiabatic_compressibility [float] adiabatic compressibility in [1/Pa].

Notes

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

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.

Returns

bulk sound velocity: float Sound velocity in [m/s].

Notes

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

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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property grueneisen_parameter
Returns the grueneisen parameter of the mineral.

Returns

gr [float] Grueneisen parameters [unitless].

Notes

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

property helmholtz
Alias for *molar_helmholtz()*

property isothermal_bulk_modulus
Returns isothermal bulk modulus of the material.

Returns

isothermal_bulk_modulus [float] Bulk modulus in [Pa].

Notes

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

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

Returns

(K_T)^-1 [float] Compressibility in [1/Pa].

Notes

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

property molar_enthalpy

Returns molar enthalpy of the mineral.

Returns

molar_enthalpy [float] Enthalpy in [J/mol].

Notes

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

property molar_entropy

Returns molar entropy of the mineral.

Returns

molar_entropy [float] Entropy in [J/K/mol].

Notes

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

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Returns

molar_gibbs [float] Gibbs free energy in [J/mol].

Notes

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

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Returns

molar_heat_capacity_p [float] Heat capacity in [J/K/mol].

Notes

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

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Returns

molar_heat_capacity_v [float] Heat capacity in [J/K/mol].

Notes

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

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Returns

molar_helmholtz [float] Helmholtz free energy in [J/mol].

Notes

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

property molar_internal_energy

Returns the molar internal energy of the mineral.

Returns

molar_internal_energy [float] The internal energy in [J/mol].

Notes

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

property molar_mass

Returns molar mass of the mineral.

Returns

molar_mass [float] Molar mass in [kg/mol].

Notes

- Needs to be implemented in derived classes.

property molar_volume

Returns molar volume of the mineral.

Returns

molar_volume [float] Molar volume in [m^3/mol].

Notes

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

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.

Returns

p_wave_velocity [float] P wave speed in [m/s].

Notes

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

property pressure

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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.

Notes

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].

property shear_modulus
Returns shear modulus of the mineral.

Returns

shear_modulus [float] Shear modulus in [Pa].

Notes

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

property shear_wave_velocity
Returns shear wave speed of the mineral.

Returns

shear_wave_velocity [float] Wave speed in [m/s].

Notes

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

property temperature

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

Returns

alpha [float] Thermal expansivity in [1/K].

Notes

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

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

name [string] Name 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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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*)
Bases: *burnman.classes.mineral.Mineral*, *burnman.classes.anisotropy.AnisotropicMaterial*

A class implementing the anisotropic mineral equation of state described in [Myh21]. This class is derived from both Mineral and AnisotropicMaterial, and inherits most of the methods from these classes.

Instantiation of an AnisotropicMineral takes three 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 a 4D array of anisotropic parameters which describe the anisotropic behaviour of the mineral. For a description of the physical meaning of these parameters, please refer to the code or to the original paper.

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.

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

deformation_gradient_tensor [2D numpy array] The deformation gradient tensor describing the deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature).

`property unrotated_cell_vectors`

Returns

unrotated_cell_vectors [2D numpy array] The vectors of the cell 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). Each vector is given in [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

`property deformed_coordinate_frame`

Returns

deformed_coordinate_frame [2D numpy array] The orientations of the three spatial coordinate axes after deformation of the mineral. For orthotropic minerals, this is equal to the identity matrix, as hydrostatic stresses only induce rotations in monoclinic and triclinic crystals.

`property rotation_matrix`

Returns

rotation_matrix [2D numpy array] 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.

`property cell_vectors`

Returns

cell_vectors [2D numpy array] The vectors of the cell constructed from one mole of formula units. Each vector is given in [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

`property cell_parameters`

Returns

cell_parameters [1D numpy array] 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.

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.

Returns

isothermal_bulk_modulus [float] Bulk modulus in [Pa].

Notes

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

property isothermal_bulk_modulus_voigt

Returns

isothermal_bulk_modulus_voigt [float] The Voigt bound on the isothermal bulk modulus in [Pa].

property isothermal_compressibility_reuss

Returns

isothermal_compressibility_reuss [float] The Reuss bound on the isothermal compressibility in [1/Pa].

property isothermal_compliance_tensor

Returns

isothermal_compliance_tensor [2D numpy array] The isothermal compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{T}pq}$).

property thermal_expansivity_tensor

Returns

thermal_expansivity_tensor [2D numpy array] The tensor of thermal expansivities [1/K].

property isothermal_stiffness_tensor

Returns

isothermak_stiffness_tensor [2D numpy array] The isothermal stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{T}pq}$).

property full_isothermal_compliance_tensor

Returns

full_isothermak_stiffness_tensor [4D numpy array] The isothermal compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{T}ijkl}$).

property full_isothermal_stiffness_tensor

Returns

full_isothermak_stiffness_tensor [4D numpy array] The isothermal stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{T}ijkl}$).

property full_isentropic_compliance_tensor

Returns

full_isothermak_stiffness_tensor [4D numpy array] The isentropic compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{N}ijkl}$).

property isentropic_compliance_tensor

Returns

isentropic_compliance_tensor [2D numpy array] The isentropic compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{N}pq}$).

property isentropic_stiffness_tensor

Returns

isentropic_stiffness_tensor [2D numpy array] The isentropic stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{N}pq}$).

property full_isentropic_stiffness_tensor

Returns

full_isentropic_stiffness_tensor [4D numpy array] The isentropic stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{N}ijkl}$).

property grueneisen_tensor

Returns

grueneisen_tensor [2D numpy array] The grueneisen tensor. This is defined by [BM67] as $\mathbb{C}_{\mathrm{N}ijkl}\alpha_{kl}V/C_P$.

property grueneisen_parameter

Anisotropic minerals do not (in general) have a single grueneisen parameter. This function returns a NotImplementedError. Users should instead consider directly querying the elements in the grueneisen_tensor.

property isothermal_compressibility_tensor

Returns

isothermal_compressibility_tensor [2D numpy array] The isothermal compressibility 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.

Returns

adiabatic_bulk_modulus [float] Adiabatic bulk modulus in [Pa].

Notes

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

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

Returns

adiabatic_compressibility [float] adiabatic compressibility in [1/Pa].

Notes

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

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.

Returns

bulk sound velocity: float Sound velocity in [m/s].

Notes

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

christoffel_tensor(*propagation_direction*)
Computes 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

copy()

debug_print(*indent=''*)
Print a human-readable representation of this Material.

property density
Returns the density of this material.

Returns

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

Notes

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

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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

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
Computes the isentropic bulk modulus (Reuss bound)

property isentropic_bulk_modulus_voigt
Computes the isentropic bulk modulus (Voigt bound)

property isentropic_bulk_modulus_vrh
Computes the isentropic bulk modulus (Voigt-Reuss-Hill average)

property isentropic_compressibility_tensor

Returns

isentropic_compressibility_tensor [2D numpy array] The isentropic compressibility tensor.

property isentropic_isotropic_poisson_ratio
Compute mu, the isotropic Poisson ratio (a description of the laterial response to loading)

isentropic_linear_compressibility(*direction*)
Computes the linear isentropic compressibility in a given direction relative to the stiffness tensor

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)
Computes the isentropic poisson ratio given loading and response directions relative to the stiffness tensor

isentropic_shear_modulus(*plane_normal*, *shear_direction*)
Computes the isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor

property isentropic_shear_modulus_reuss
Computes the isentropic shear modulus (Reuss bound)

property isentropic_shear_modulus_voigt
Computes the isentropic shear modulus (Voigt bound)

property isentropic_shear_modulus_vrh
Computes the shear modulus (Voigt-Reuss-Hill average)

property isentropic_universal_elastic_anisotropy
Compute the universal elastic anisotropy

isentropic_youngs_modulus(*direction*)
Computes the isentropic Youngs modulus in a given direction relative to the stiffness tensor

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

Returns

(K_T)^-1 [float] Compressibility in [1/Pa].

Notes

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

property molar_enthalpy

Returns molar enthalpy of the mineral.

Returns

molar_enthalpy [float] Enthalpy in [J/mol].

Notes

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

property molar_entropy

Returns molar entropy of the mineral.

Returns

molar_entropy [float] Entropy in [J/K/mol].

Notes

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

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

Returns

molar_gibbs [float] Gibbs free energy in [J/mol].

Notes

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

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

Returns

molar_heat_capacity_p [float] Heat capacity in [J/K/mol].

Notes

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

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

Returns

molar_heat_capacity_v [float] Heat capacity in [J/K/mol].

Notes

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

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

Returns

molar_helmholtz [float] Helmholtz free energy in [J/mol].

Notes

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

property molar_internal_energy

Returns the molar internal energy of the mineral.

Returns

molar_internal_energy [float] The internal energy in [J/mol].

Notes

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

property molar_mass

Returns molar mass of the mineral.

Returns

molar_mass [float] Molar mass in [kg/mol].

Notes

- Needs to be implemented in derived classes.

property molar_volume
Returns molar volume of the mineral.

Returns

molar_volume [float] Molar volume in [m^3/mol].

Notes

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

property p_wave_velocity
Returns P wave speed of the mineral.

Returns

p_wave_velocity [float] P wave speed in [m/s].

Notes

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

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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.

property shear_wave_velocity
Returns shear wave speed of the mineral.

Returns

shear_wave_velocity [float] Wave speed in [m/s].

Notes

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

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

property thermal_expansivity
Returns thermal expansion coefficient of the mineral.

Returns

alpha [float] Thermal expansivity in [1/K].

Notes

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

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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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*)
Computes the compressional wave velocity, and two shear wave velocities in a given propagation direction

Returns two lists, containing the wave speeds and directions of particle motion relative to the stiffness tensor

burnman.**cell_parameters_to_vectors**(*cell_parameters*)
Converts cell parameters to unit cell vectors.

Parameters

cell_parameters [1D numpy array] 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

M [2D numpy array] 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.

burnman.**cell_vectors_to_parameters**(*M*)
Converts unit cell vectors to cell parameters.

Parameters

M [2D numpy array] 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

cell_parameters [1D numpy array] 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.

5.1.4 Composites

class burnman.**Composite**(*phases*, *fractions=None*, *fraction_type='molar'*, *name='Unnamed composite'*)

Bases: *burnman.classes.material.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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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.

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 SolidSolution.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 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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property helmholtz
Alias for *molar_helmholtz()*

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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()*

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

5.2 Equations of state

5.2.1 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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

volume [float] Molar volume of the mineral. $[m^3]$

`pressure`(*temperature*, *volume*, *params*)

Parameters

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [m^3]

`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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

gamma [float] Grueneisen parameter of the mineral. $[unitless]$

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

K_T [float] Isothermal bulk modulus of the mineral. $[Pa]$

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

K_S [float] Adiabatic bulk modulus of the mineral. $[Pa]$

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

G [float] Shear modulus of the mineral. $[Pa]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

C_V [float] Heat capacity at constant volume of the mineral. $[J/K/mol]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

C_P [float] Heat capacity at constant pressure of the mineral. $[J/K/mol]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

alpha [float] Thermal expansivity of the mineral. $[1/K]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

G [float] Gibbs free energy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

entropy(*pressure*, *temperature*, *volume*, *params*)
Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

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 [dictionary] Dictionary containing material parameters required by the equation of state.

5.2.2 Murnaghan

class burnman.eos.**Murnaghan**

Bases: *burnman.eos.equation_of_state.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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [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. $[Pa]$ 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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.3 Birch-Murnaghan

5.2.3.1 Base class

class burnman.eos.birch_murnaghan.**BirchMurnaghanBase**

Bases: *burnman.eos.equation_of_state.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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [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. $[J/K/mol]$

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.3.2 BM2

class burnman.eos.**BM2**

Bases: *burnman.eos.birch_murnaghan.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

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

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [m^3]

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 $[Pa]$.

5.2.3.3 BM3

class burnman.eos.**BM3**
Bases: *burnman.eos.birch_murnaghan.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

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

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [m^3]

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 $[Pa]$.

5.2.3.4 BM4

class burnman.eos.**BM4**
Bases: *burnman.eos.equation_of_state.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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [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. $[J/K/mol]$

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.4 Vinet

class burnman.eos.**Vinet**

Bases: *burnman.eos.equation_of_state.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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [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. $[Pa]$ 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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.5 Morse Potential

class burnman.eos.**Morse**

Bases: *burnman.eos.equation_of_state.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

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [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. $[J/K/mol]$

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.6 Reciprocal K-prime

class burnman.eos.**RKprime**

Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.7 Stixrude and Lithgow-Bertelloni Formulation

5.2.7.1 Base class

class burnman.eos.slb.**SLBBase**

Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

5.2.7.2 SLB2

class burnman.eos.**SLB2**

Bases: *burnman.eos.slb.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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.

5.2.7.3 SLB3

class burnman.eos.**SLB3**
Bases: *burnman.eos.slb.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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.

5.2.8 Mie-Grüneisen-Debye

5.2.8.1 Base class

class burnman.eos.mie_grueneisen_debye.**MGDBase**

Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

5.2.8.2 MGD2

class burnman.eos.**MGD2**
Bases: *burnman.eos.mie_grueneisen_debye.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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

5.2.8.3 MGD3

class burnman.eos.**MGD3**

Bases: *burnman.eos.mie_grueneisen_debye.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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

5.2.9 Modified Tait

class burnman.eos.**MT**
Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

helmholtz_free_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

5.2.10 Holland and Powell Formulations

5.2.10.1 HP_TMT (2011 solid formulation)

class burnman.eos.**HP_TMT**

Bases: *burnman.eos.equation_of_state.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

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

5.2.10.2 HP_TMTL (2011 liquid formulation)

class burnman.eos.**HP_TMTL**

Bases: *burnman.eos.equation_of_state.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

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

5.2.10.3 HP98 (1998 formulation)

class burnman.eos.**HP98**

Bases: *burnman.eos.equation_of_state.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

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

5.2.11 De Koker Solid and Liquid Formulations

5.2.11.1 DKS_S (Solid formulation)

class burnman.eos.**DKS_S**

Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

5.2.11.2 DKS_L (Liquid formulation)

class burnman.eos.**DKS_L**
Bases: *burnman.eos.equation_of_state.EquationOfState*

Base class for the finite strain liquid equation of state detailed in [deKokerKarkiStixrude13] (supplementary materials).

pressure(*temperature*, *volume*, *params*)

Parameters

volume [float] Molar volume at which to evaluate the equation of state. [m^3]

temperature [float] Temperature at which to evaluate the equation of state. [K]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

pressure [float] Pressure of the mineral, including cold and thermal parts. [m^3]

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

volume [float] Molar volume of the mineral. $[m^3]$

`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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

5.2.12 Anderson and Ahrens (1994)

class burnman.eos.**AA**
Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

5.2.13 CoRK

class burnman.eos.**CORK**
Bases: *burnman.eos.equation_of_state.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 :func:`volume`. :math:`[m^3]`

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

density [float] Density of the mineral. $[kg/m^3]$

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]

params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

H [float] Enthalpy of the mineral

`entropy`(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

`helmholtz_free_energy`(*pressure*, *temperature*, *volume*, *params*)

Parameters

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 [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

F [float] Helmholtz free energy of the mineral

`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]

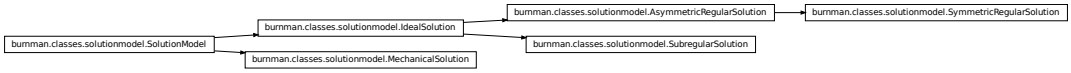
params [dictionary] Dictionary containing material parameters required by the equation of state.

Returns

U [float] Internal energy of the mineral

5.3 Solution models

SolidSolution objects in Burnman (type *SolidSolution*) take one of several methods which define the properties of the solution.



5.3.1 Base class

class burnman.**SolidSolution**(*name=None*, *solution_type=None*, *endmembers=None*, *energy_interaction=None*, *volume_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *volume_ternary_terms=None*, *entropy_ternary_terms=None*, *alphas=None*, *molar_fractions=None*)

Bases: *burnman.classes.mineral.Mineral*

This is the base class for all solid 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 solid solution can only be queried after setting the pressure, temperature and composition using set_state().

This class is available as *burnman.SolidSolution*. It uses an instance of *burnman.SolutionModel* to calculate interaction terms between endmembers.

All the solid 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 solid solution

solution_type [string] String determining which SolutionModel to use. One of 'mechanical', 'ideal', 'symmetric', 'asymmetric' or 'subregular'.

endmembers [list of lists] List of endmembers in this solid solution. The first item of each list should be a *burnman.Mineral* object. The second item should be a string with the site formula of the endmember.

molar_fractions [numpy array (optional)] The molar fractions of each endmember in the solid 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.

get_endmembers()

set_composition(*molar_fractions*)
Set the composition for this solid 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 solid 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 solid solutions.

property excess_partial_volumes
Returns excess partial volumes [m^3]. Property specific to solid solutions.

property excess_partial_entropies
Returns excess partial entropies [J/K]. Property specific to solid solutions.

property partial_gibbs
Returns excess partial molar gibbs free energy [J/mol]. Property specific to solid solutions.

property partial_volumes
Returns excess partial volumes [m^3]. Property specific to solid solutions.

property partial_entropies
Returns excess partial entropies [J/K]. Property specific to solid solutions.

property excess_gibbs
Returns molar excess gibbs free energy [J/mol]. Property specific to solid solutions.

property gibbs_hessian
Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solid solutions.

property entropy_hessian
Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solid solutions.

property volume_hessian
Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solid solutions.

property molar_gibbs
Returns molar Gibbs free energy of the solid solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz
Returns molar Helmholtz free energy of the solid solution [J/mol]. Aliased with self.helmholtz.

property molar_mass
Returns molar mass of the solid solution [kg/mol].

property excess_volume
Returns excess molar volume of the solid solution [m^3/mol]. Specific property for solid solutions.

property molar_volume
Returns molar volume of the solid solution [m^3/mol]. Aliased with self.V.

property density
Returns density of the solid solution [kg/m^3]. Aliased with self.rho.

property excess_entropy
Returns excess molar entropy [J/K/mol]. Property specific to solid solutions.

property molar_entropy
Returns molar entropy of the solid solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy
Returns excess molar enthalpy [J/mol]. Property specific to solid solutions.

property molar_enthalpy
Returns molar enthalpy of the solid solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus
Returns isothermal bulk modulus of the solid solution [Pa]. Aliased with self.K_T.

property adiabatic_bulk_modulus
Returns adiabatic bulk modulus of the solid solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility
Returns isothermal compressibility of the solid solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property adiabatic_compressibility
Returns adiabatic compressibility of the solid solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus
Returns shear modulus of the solid solution [Pa]. Aliased with self.G.

property p_wave_velocity
Returns P wave speed of the solid solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity
Returns bulk sound speed of the solid solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity
Returns shear wave speed of the solid solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter
Returns grueneisen parameter of the solid solution [unitless]. Aliased with self.gr.

property thermal_expansivity
Returns thermal expansion coefficient (alpha) of the solid solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v
Returns molar heat capacity at constant volume of the solid solution [J/K/mol]. Aliased with self.C_v.

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 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 [ndlist or ndarray of float] n-dimensional array of pressures in [Pa].

temperatures [ndlist or ndarray of float] n-dimensional array of temperatures in [K].

Returns

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

property gibbs
Alias for *molar_gibbs()*

property gr
Alias for *grueneisen_parameter()*

property helmholtz
Alias for *molar_helmholtz()*

property molar_heat_capacity_p
Returns molar heat capacity at constant pressure of the solid solution [J/K/mol]. Aliased with self.C_p.

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

Returns

pressure [float] Pressure in [Pa].

Notes

- Aliased with *P()*.

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()*

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

Returns

temperature [float] Temperature in [K].

Notes

- Aliased with *T()*.

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).

Returns

fractions [list of float] List of molar fractions, should sum to 1.0.

minerals [list of *burnman.Mineral*] List of minerals.

Notes

Needs to be implemented in derived classes.

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 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 solid solution.

property endmember_formulae
A list of formulae for all the endmember in the solid solution.

property endmember_names
A list of names for all the endmember in the solid solution.

property n_endmembers
The number of endmembers in the solid solution.

property elements
A list of the elements which could be contained in the solid solution, returned in the IUPAC element order.

class burnman.**SolutionModel**
Bases: object

This is the base class for a solution model, intended for use in defining solid solutions and performing thermodynamic calculations on them. All minerals of type *burnman.SolidSolution* use a solution model for defining how the endmembers in the solid 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 Gibbs free energy and molar volume of a solid solution will be equal to the weighted arithmetic averages of the different endmember values.

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

5.3.2 Mechanical solution

`class burnman.classes.solutionmodel.MechanicalSolution(endmembers)`

Bases: *burnman.classes.solutionmodel.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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

`activity_coefficients`(*pressure*, *temperature*, *molar_fractions*)

`activities`(*pressure*, *temperature*, *molar_fractions*)

5.3.3 Ideal solution

`class burnman.classes.solutionmodel.IdealSolution`(*endmembers*)

Bases: *burnman.classes.solutionmodel.SolutionModel*

A very simple class representing an ideal solution model. Calculate the excess gibbs free energy and entropy due to configurational entropy, excess volume is equal to zero.

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

`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*)

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

5.3.4 Asymmetric regular solution

class burnman.classes.solutionmodel.**AsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *burnman.classes.solutionmodel.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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

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*)

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

5.3.5 Symmetric regular solution

class burnman.classes.solutionmodel.**SymmetricRegularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *burnman.classes.solutionmodel.AsymmetricRegularSolution*

Solution model implementing the symmetric regular solution model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class.

activities(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

5.3.6 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: *burnman.classes.solutionmodel.IdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess conconfigurational 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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_G_excess [numpy array] The excess Gibbs free energy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_S_excess [numpy array] The excess entropy of each endmember

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

partial_V_excess [numpy array] The excess volume of each endmember

`gibbs_hessian`(*pressure*, *temperature*, *molar_fractions*)

`entropy_hessian`(*pressure*, *temperature*, *molar_fractions*)

`volume_hessian`(*pressure*, *temperature*, *molar_fractions*)

`activity_coefficients`(*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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

H_excess [float] The excess enthalpy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

S_excess [float] The excess entropy of the solution

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

G_excess [float] The excess Gibbs free energy

`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. [K]

molar_fractions [list of floats] List of molar fractions of the different endmembers in solution

Returns

V_excess [float] The excess volume of the solution

`activities`(*pressure*, *temperature*, *molar_fractions*)

5.4 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 SolidSolution object.

Parameters

solution [*burnman.SolidSolution* 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 [string (optional)] A name corresponding to the new solution

endmember_names [list of strings (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

solution [*burnman.SolidSolution* object] The transformed solid solution

5.5 Compositions

5.5.1 Base class

class burnman.**Composition**(*composition_dictionary*, *unit_type='weight'*, *normalize=False*)

Bases: object

Class for a composition object, which can be used to store, modify and renormalize compositions, and also convert between molar, weight, and atomic amounts.

This class is available as burnman.Composition.

renormalize(*unit_type*, *normalization_component*, *normalization_amount*)

Change the normalization for a given unit type (weight, molar, or atomic) Resets cached composition only for that unit type

Parameters

unit_type ['weight', 'molar' or 'atomic'] Unit type composition to be renormalised

normalization_component: string Component/element on which to renormalize. String must either be one of the components/elements already in composite, 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 (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, in dictionary form

unit_type ['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 molar_composition`
Returns the molar composition as a counter [moles]

`property weight_composition`
Returns the weight composition as a counter [g]

`property atomic_composition`
Returns the atomic composition as a counter [moles]

`print`(*unit_type*, *significant_figures=1*, *normalization_component='total'*, *normalization_amount=100.0*)
Pretty-print function for the composition This does not renormalize the Composition internally

Parameters

unit_type ['weight', 'molar' or 'atomic'] Unit type in which to print the composition

significant_figures [integer] Number of significant figures for each amount

normalization_component: string 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. (default = '100.')

5.5.2 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 [string] 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 ['weight' or 'molar'] Specify whether the compositions in the file are given as weight or molar amounts.

normalize [boolean] 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).

5.5.3 Fitting functions

burnman.optimize.composition_fitting.**fit_composition_to_solution**(*solution*, *fitted_variables*, *variable_values*, *variable_covariances*, *variable_conversions=None*, *normalize=True*)

Takes a SolidSolution 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.SolidSolution object] The solution to use in the fitting procedure.

fitted_variables [list of strings] 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 [dictionary of dictionaries 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 [boolean (default: True)] If True, normalizes the optimized molar fractions to sum to unity.

Returns

popt [numpy array] Optimized molar fractions.

pcov [2D numpy array] Covariance matrix corresponding to the optimized molar fractions.

res [float] The weighted residual of the fitting procedure.

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

popt [numpy array] Optimized phase amounts.

pcov [2D numpy array] Covariance matrix corresponding to the optimized phase amounts.

res [float] The weighted residual of the fitting procedure.

5.6 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.

5.6.1 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 [boolean (default is False)] 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 [integer (default is 2)] Number of points per edge of the polytope.

unique_sorted [boolean (default is True)] 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 ['independent endmember proportions' (default) or 'site occupancies'] Whether to grid the polytope in terms of independent endmember proportions or site occupancies.

limits [2D numpy array] Additional inequalities restricting the gridded area of the polytope.

Returns

——-

points [2D numpy array] A list of points gridding the polytope.

5.6.2 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 [boolean] Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns

polytope [`burnman.polytope.MaterialPolytope` object] A polytope object corresponding to the parameters provided.

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 [boolean] Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns

polytope [burnman.polytope.MaterialPolytope object] A polytope object corresponding to the parameters provided.

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 SolidSolution and Mineral objects.

composition [dictionary] A dictionary containing the amounts of each element.

return_fractions [boolean] Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns

polytope [burnman.polytope.MaterialPolytope object] A polytope object corresponding to the parameters provided.

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 solid solutions, this function will return a composite that only contains the Mg-bearing endmembers.

Parameters

composite [*burnman.Composite* object] The initial Composite object

composition [dictionary] A dictionary containing the amounts of each element

Returns

simple_composite [*burnman.Composite* object] The simplified Composite object

5.7 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.

5.7.1 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. $[m^3]$

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

K [float] The average bulk modulus K. $[Pa]$

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

G [float] The average shear modulus G. $[Pa]$

`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

rho [float] Density ρ. $[kg/m^3]$

`average_thermal_expansivity`(*volumes*, *alphas*)

thermal expansion coefficient of the mineral α. $[1/K]$

`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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

`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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

5.7.2 Voigt bound

class burnman.averaging_schemes.**Voigt**

Bases: *burnman.classes.averaging_schemes.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. Not used in this average. $[Pa]$

Returns

K [float] The Voigt average bulk modulus K_V. $[Pa]$

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. Not used in this average. $[Pa]$

shear_moduli [list of floats] List of shear moduli G of each phase in the composite. $[Pa]$

Returns

G [float] The Voigt average shear modulus G_V. $[Pa]$

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

rho [float] Density ρ. $[kg/m^3]$

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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

average_thermal_expansivity(*volumes*, *alphas*)

thermal expansion coefficient of the mineral α. $[1/K]$

5.7.3 Reuss bound

class burnman.averaging_schemes.**Reuss**

Bases: *burnman.classes.averaging_schemes.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. Not used in this average. $[Pa]$

Returns

K [float] The Reuss average bulk modulus K_R. $[Pa]$

`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. Not used in this average. $[Pa]$

shear_moduli [list of floats] List of shear moduli G of each phase in the composite. $[Pa]$

Returns

G [float] The Reuss average shear modulus G_R. $[Pa]$

`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

rho [float] Density ρ. $[kg/m^3]$

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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

average_thermal_expansivity(*volumes*, *alphas*)
thermal expansion coefficient of the mineral α. $[1/K]$

5.7.4 Voigt-Reuss-Hill average

class burnman.averaging_schemes.**VoigtReussHill**
Bases: *burnman.classes.averaging_schemes.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. Not used in this average. $[Pa]$

Returns

K [float] The Voigt-Reuss-Hill average bulk modulus K_{VRH}. $[Pa]$

`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`.

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 Not used in this average. $[Pa]$

shear_moduli [list of floats] List of shear moduli G of each phase in the composite $[Pa]$

Returns

G [float] The Voigt-Reuss-Hill average shear modulus G_{VRH}. $[Pa]$

`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

rho [float] Density ρ. $[kg/m^3]$

`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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

average_thermal_expansivity(*volumes*, *alphas*)

thermal expansion coefficient of the mineral α. $[1/K]$

5.7.5 Hashin-Shtrikman upper bound

class burnman.averaging_schemes.**HashinShtrikmanUpper**

Bases: *burnman.classes.averaging_schemes.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 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. $[Pa]$

Returns

K [float] The upper Hashin-Shtrikman average bulk modulus K. $[Pa]$

`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 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. $[Pa]$

Returns

G [float] The upper Hashin-Shtrikman average shear modulus G. $[Pa]$

`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

rho [float] Density ρ. $[kg/m^3]$

`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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

`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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

average_thermal_expansivity(*volumes*, *alphas*)
thermal expansion coefficient of the mineral α. $[1/K]$

5.7.6 Hashin-Shtrikman lower bound

class burnman.averaging_schemes.**HashinShtrikmanLower**
Bases: *burnman.classes.averaging_schemes.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 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. $[Pa]$

Returns

K [float] The lower Hashin-Shtrikman average bulk modulus K. $[Pa]$

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 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

G [float] The lower Hashin-Shtrikman average shear modulus G. $[Pa]$

`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

rho [float] Density ρ. $[kg/m^3]$

`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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

`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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

`average_thermal_expansivity`(*volumes*, *alphas*)

thermal expansion coefficient of the mineral α. $[1/K]$

5.7.7 Hashin-Shtrikman arithmetic average

class burnman.averaging_schemes.**HashinShtrikmanAverage**

Bases: *burnman.classes.averaging_schemes.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

K [float] The arithmetic mean of the Hashin-Shtrikman bounds on bulk modulus K. $[Pa]$

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

G [float] The arithmetic mean of the Hashin-Shtrikman bounds on shear modulus G. $[Pa]$

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

rho [float] Density ρ. $[kg/m^3]$

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

c_p [float] heat capacity at constant pressure C_P of the composite. $[J/K/mol]$

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

c_v [float] heat capacity at constant volume of the composite C_V. $[J/K/mol]$

average_thermal_expansivity(*volumes*, *alphas*)

thermal expansion coefficient of the mineral α. $[1/K]$

5.8 Geotherms

5.9 Layers and Planets

5.9.1 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 for an adiabat.

`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 (default = 50)

max_delta [float] Relative update to the highest pressure in the layer between iterations to stop iterations (default = 1.e-5)

`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

property_array [numpy array] 1D or 2D array of requested properties (1D if only one property was requested)

`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

pressure [array of floats] Pressures in [Pa] at the predefined radii.

property temperature

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

- Aliased with *T()*.

Returns

temperature [array of floats] Temperatures in [K] at the predefined radii.

property molar_internal_energy

Returns the molar internal energies across the layer.

Returns

molar_internal_energy [array of floats] The internal energies in [J/mol] at the predefined radii.

Notes

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

property molar_gibbs

Returns the molar Gibbs free energies across the layer.

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

Returns

molar_gibbs [array of floats] Gibbs free energies in [J/mol] at the predefined radii.

property molar_helmholtz

Returns the molar Helmholtz free energies across the layer.

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

Returns

molar_helmholtz [array of floats] Helmholtz free energies in [J/mol] at the predefined radii.

property molar_mass

Returns molar mass of the layer.

Needs to be implemented in derived classes.

Returns

molar_mass [array of floats] Molar mass in [kg/mol].

property molar_volume
Returns molar volumes across the layer.

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

Returns

molar_volume [array of floats] Molar volumes in [m^3/mol] at the predefined radii.

property density
Returns the densities across this layer.

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

Returns

density [array of floats] The densities of this material in [kg/m^3] at the predefined radii.

property molar_entropy
Returns molar entropies acroos the layer.

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

Returns

molar_entropy [array of floats] Entropies in [J/K/mol] at the predefined radii.

property molar_enthalpy
Returns molar enthalpies across the layer.

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

Returns

molar_enthalpy [array of floats] Enthalpies in [J/mol] at the predefined radii.

property isothermal_bulk_modulus
Returns isothermal bulk moduli across the layer.

Returns

isothermal_bulk_modulus [array of floats] Bulk moduli in [Pa] at the predefined radii.

Notes

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

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 [array of floats] Adiabatic bulk modulus in [Pa] at the predefined radii.

`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

(K_T)^-1 [array of floats] Compressibilities in [1/Pa] at the predefined radii.

`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_compressibility [array of floats] adiabatic compressibilities in [1/Pa] at the predefined radii.

`property shear_modulus`

Returns shear moduli across the layer.

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

Returns

shear_modulus [array of floats] Shear modulie in [Pa] at the predefined radii.

`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_velocity [array of floats] P wave speeds in [m/s] at the predefined radii.

`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 velocity: array of floats Sound velocities in [m/s] at the predefined radii.

`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_velocity [array of floats] Wave speeds in [m/s] at the predefined radii.

property grueneisen_parameter
Returns the grueneisen parameters across the layer.

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

Returns

gr [array of floats] Grueneisen parameters [unitless] at the predefined radii.

property thermal_expansivity
Returns thermal expansion coefficients across the layer.

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

Returns

alpha [array of floats] Thermal expansivities in [1/K] at the predefined radii.

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

molar_heat_capacity_v [array of floats] Heat capacities in [J/K/mol] at the predefined radii.

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

molar_heat_capacity_p [array of floats] Heat capacities in [J/K/mol] at the predefined radii.

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()*

5.9.2 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 [string] Given name of a layer

`get_layer_by_radius(`*radius*`)`
Returns a layer in which this radius lies

Parameters

radius [float] radius at which to evaluate the 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

properties_array [numpy array] 1D or 2D array of requested properties (1D if only one property was requested)

`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 [string] 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 (default = 50)

`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 [array of floats] Pressure in [Pa].

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

Aliased with *T()*.

Returns

temperature [array of floats] Temperature in [K].

property molar_internal_energy
Returns the molar internal energy of the planet.

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

Returns

molar_internal_energy [array of floats] The internal energy in [J/mol].

property molar_gibbs
Returns the molar Gibbs free energy of the planet.

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

Returns

molar_gibbs [array of floats] Gibbs free energy in [J/mol].

property molar_helmholtz
Returns the molar Helmholtz free energy of the planet.

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

Returns

molar_helmholtz [array of floats] Helmholtz free energy in [J/mol].

property molar_mass
Returns molar mass of the planet.

Needs to be implemented in derived classes.

Returns

molar_mass [array of floats] Molar mass in [kg/mol].

property molar_volume
Returns molar volume of the planet.

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

Returns

molar_volume [array of floats] Molar volume in [m^3/mol].

property density
Returns the density of this planet.

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

Returns

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

property molar_entropy
Returns molar entropy of the planet.

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

Returns

molar_entropy [array of floats] Entropy in [J/mol].

property molar_enthalpy
Returns molar enthalpy of the planet.

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

Returns

molar_enthalpy [array of floats] Enthalpy in [J/mol].

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 [array of floats] Bulk modulus in [Pa].

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 [array of floats] Adiabatic bulk modulus in [Pa].

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

(K_T)^-1 [array of floats] Compressibility in [1/Pa].

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 [array of floats] adiabatic compressibility in [1/Pa].

property shear_modulus
Returns shear modulus of the planet.

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

Returns

shear_modulus [array of floats] Shear modulus in [Pa].

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_velocity [array of floats] P wave speed in [m/s].

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: array of floats Sound velocity in [m/s].

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_velocity [array of floats] Wave speed in [m/s].

property grueneisen_parameter
Returns the grueneisen parameter of the planet.

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

Returns

gr [array of floats] Grueneisen parameters [unitless].

property thermal_expansivity
Returns thermal expansion coefficient of the planet.

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

Returns

alpha [array of floats] Thermal expansivity in [1/K].

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

molar_heat_capacity_v [array of floats] Heat capacity in [J/K/mol].

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

molar_heat_capacity_p [array of floats] Heat capacity in [J/K/mol].

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()*

5.10 Thermodynamics

Burnman has a number of functions and classes which deal with the thermodynamics of single phases and aggregates.

5.10.1 Lattice Vibrations

5.10.1.1 Debye model

burnman.eos.debye.**jit**(*fn*)

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. It is important to note that this does NOT include the zero point energy of vibration for the lattice. As long as you are calculating relative differences in F, this should cancel anyways. In Joules.

burnman.eos.debye.**entropy**(*T*, *debye_T*, *n*)
Entropy due to lattice vibrations in the Debye model [J/K]

5.10.1.2 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

5.10.2 Chemistry parsing and thermodynamics

burnman.tools.chemistry.**read_masses**()
A simple function to read a file with a two column list of elements and their masses into a dictionary

burnman.tools.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.tools.chemistry.**dictionarize_formula**(*formula*)

A function to read a chemical formula string and convert it into a dictionary

Parameters

formula [string object] Chemical formula, written in the XnYm format, where the formula has n atoms of element X and m atoms of element Y

Returns

f [dictionary object] The same chemical formula, but expressed as a dictionary.

burnman.tools.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

summed_formula [Counter object] The sum of the user-provided formulae

burnman.tools.chemistry.**formula_mass**(*formula*)

A function to take a chemical formula and compute the formula mass.

Parameters

formula [dictionary or counter object] A chemical formula

Returns

mass [float] The mass per mole of formula

burnman.tools.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 [dictionary or counter object] A chemical formula

to_type [string, one of 'mass' or 'molar'] Conversion type

normalize [boolean] Whether or not to normalize the converted formula to 1

Returns

f [dictionary] The converted formula

burnman.tools.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 [instance of class] Class must have a "formulas" attribute, containing a list of chemical formulae with site information

Returns

none Nothing is returned from this function, but the solution_model object gains the following attributes.

solution_formulae [list of dictionaries] List of endmember formulae is output from site formula strings

n_sites [integer] Number of sites in the solid solution. Should be the same for all endmembers.

sites [list of lists of strings] A list of elements for each site in the solid solution

site_names [list of strings] A list of elements_site pairs in the solid 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 elements on each of the sites in the solid solution. Example: A binary solution [[A][B],[B][C1/2D1/2]] would have n_occupancies = 5, with two possible elements on Site 1 and three on Site 2

site_multiplicities [array of floats] The number of each site per formula unit To simplify computations later, the multiplicities are repeated for each element on each site

endmember_occupancies [2d array of floats] A 1D array for each endmember in the solid solution, containing the fraction of atoms of each element on each site.

endmember_noccupancies [2d array of floats] A 1D array for each endmember in the solid solution, containing the number of atoms of each element on each site per mole of endmember.

burnman.tools.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 [numpy array of floats] List of floats giving the multiplicity of each site 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

site_formulae [list of strings] A list of strings in standard burnman format. For example, [Mg]3[Al]2 would correspond to the classic two-site pyrope garnet.

burnman.tools.chemistry.**compositional_array**(*formulae*)

Parameters

formulae [list of dictionaries] List of chemical formulae

Returns

formula_array [2D array of floats] Array of endmember formulae

elements [List of strings] List of elements

burnman.tools.chemistry.**ordered_compositional_array**(*formulae*, *elements*)

Parameters

formulae [list of dictionaries] List of chemical formulae

elements [List of strings] List of elements

Returns

formula_array [2D array of floats] Array of endmember formulae

burnman.tools.chemistry.**formula_to_string**(*formula*)

Parameters

formula [dictionary or counter] Chemical formula

Returns

formula_string [string] 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.

burnman.tools.chemistry.**sort_element_list_to_IUPAC_order**(*element_list*)

Parameters

element_list [list] List of elements

Returns

sorted_list [list] List of elements sorted into IUPAC order

burnman.tools.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 [string] Input fraction type: 'molar', 'mass' or 'volume'

output_type [string] Output fraction type: 'molar', 'mass' or 'volume'

Returns

output_fractions [list of floats] List of output phase fractions (of type output_type)

burnman.tools.chemistry.**chemical_potentials**(*assemblage*, *component_formulae*)

The compositional space of the components does not have to be a superset of the compositional space of the assemblage. Nor do they have to compose an orthogonal basis.

The components must each be described by a linear mineral combination

The mineral compositions must be linearly independent

Parameters

assemblage [list of classes]

List of material classes set_method and set_state should already have been used the composition of the solid solutions should also have been set

component_formulae [list of dictionaries] List of chemical component formula dictionaries No restriction on length

Returns

component_potentials [array of floats] Array of chemical potentials of components

burnman.tools.chemistry.**fugacity**(*standard_material*, *assemblage*)

Parameters

standard_material: class Material class set_method and set_state should already have been used material must have a formula as a dictionary parameter

assemblage: list of classes List of material classes set_method and set_state should already have been used

Returns

fugacity [float] Value of the fugacity of the component with respect to the standard material

burnman.tools.chemistry.**relative_fugacity**(*standard_material*, *assemblage*, *reference_assemblage*)

Parameters

standard_material: class Material class set_method and set_state should already have been used material must have a formula as a dictionary parameter

assemblage: list of classes List of material classes set_method and set_state should already have been used

reference_assemblage: list of classes List of material classes set_method and set_state should already have been used

Returns

relative_fugacity [float] Value of the fugacity of the component in the assemblage with respect to the reference_assemblage

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 minerals] 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 [optional float] Initial pressure guess [Pa]

Returns

pressure [float] The equilibrium pressure of the reaction [Pa]

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 minerals] 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 [optional float] Initial temperature guess [K]

Returns

temperature [float] The equilibrium temperature of the reaction [K]

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 minerals] 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 [optional float] Initial temperature guess [K]

Returns

temperature [float] The equilibrium temperature of the reaction [K]

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 [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 [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

temperatures [numpy array of floats] The Hugoniot temperatures at pressure

volumes [numpy array of floats] The Hugoniot volumes at pressure

5.11 Equilibrium Thermodynamics

burnman.**equilibrate**(*composition*, *assemblage*, *equality_constraints*, *tol=0.001*, *store_iterates=False*, *store_assemblage=True*, *max_iterations=100.0*, *verbose=False*)

A function that equilibrates an assemblage subject to given bulk composition and equality constraints by solving the equilibrium relations (chemical affinities for feasible reactions in the system should be equal to zero).

Parameters

composition [dictionary] The bulk composition that the assemblage must satisfy

assemblage [burnman.Composite object] The assemblage to be equilibrated

equality_constraints [list] A list of equality constraints. 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 <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 n*x/d*x = 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.

tol [float] The tolerance for the nonlinear solver.

store_iterates [boolean] Whether to store the parameter values for each iteration in each solution object.

store_assemblage [boolean] Whether to store a copy of the assemblage object in each solution object.

max_iterations [integer] The maximum number of iterations for the nonlinear solver.

verbose [boolean] Whether to print output updating the user on the status of equilibration.

Returns

sol_list [single, list, or 2D list of solver solution objects]

prm [namedtuple object] 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).

5.12 Seismic

5.12.1 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','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)).

internal_depth_list(*mindepth=0.0*, *maxdepth=1e+99*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

`density`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

`G`(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

`K`(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

`QK`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

`QG`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

`depth`(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] Depth(s) [m] for given pressure(s)

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

5.12.2 Class for 1D Models

class burnman.classes.seismic.**SeismicTable**

Bases: *burnman.classes.seismic.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 where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

pressure(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

gravity(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

v_p(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

`QK`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

`QG`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

`density`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`depth`(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] Depth(s) [m] for given pressure(s)

`radius`(*pressure*)

`G`(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

`K`(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

`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','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)).

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

5.12.3 Models currently implemented

`class` burnman.classes.seismic.`PREM`

Bases: *burnman.classes.seismic.SeismicTable*

Reads PREM (1s) (input_seismic/prem.txt, [DA81]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

K(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

QG(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

QK(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

depth(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`radius`(*pressure*)

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

`class` burnman.classes.seismic.`Slow`

Bases: *burnman.classes.seismic.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 array of floats] Shear modulus at given for depth(s) in [Pa].

`K`(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

`QG`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

`QK`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`density`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

`depth`(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`radius`(*pressure*)

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

class burnman.classes.seismic.**Fast**

Bases: *burnman.classes.seismic.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 array of floats] Shear modulus at given for depth(s) in [Pa].

K(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

QG(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

QK(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

`depth`(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

radius(*pressure*)

v_p(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

v_phi(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

v_s(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

class burnman.classes.seismic.**STW105**

Bases: *burnman.classes.seismic.SeismicTable*

Reads STW05 (a.k.a. REF) (1s) (input_seismic/STW105.txt, [KED08]). See also burnman.seismic.SeismicTable.

G(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

K(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

QG(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

QK(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

depth(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`radius`(*pressure*)

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

`class burnman.classes.seismic.IASP91`

Bases: *burnman.classes.seismic.SeismicTable*

Reads REF/STW05 (input_seismic/STW105.txt, [KED08]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

`K`(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

`QG`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

`QK`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

bullen(*depth*)
Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

depth(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

gravity(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)
Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`radius`(*pressure*)

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

`v_phi`(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

`v_s`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

`class burnman.classes.seismic.AK135`

Bases: *burnman.classes.seismic.SeismicTable*

Reads AK135 (input_seismic/ak135.txt, [KEB95]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters

depth [float or array of floats] Shear modulus at given for depth(s) in [Pa].

`K`(*depth*)

Parameters

depth [float or array of floats] Bulk modulus at given for depth(s) in [Pa]

`QG`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

QG [float or array of floats] Quality factor (dimensionless) for shear modulus at given depth(s).

`QK`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

Qk [float or array of floats] Quality factor (dimensionless) for bulk modulus at given depth(s).

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`density`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

density [float or array of floats] Density at given depth(s) in [kg/m^3].

`depth`(*pressure*)

Parameters

pressure [float or array of floats] Pressure(s) [Pa] to evaluate depth at.

Returns

depth [float or array of floats] 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','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)).

`gravity`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate gravity at.

Returns

gravity [float or array of floats] Gravity for given depths in [m/s^2]

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths where this seismic data is specified at. 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 Maximum depth value to be returned [m]

discontinuity interval Shift continuities to remove ambigious values for depth, default value = 1 [m]

Returns

depths [array of floats] Depths [m].

`pressure`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

pressure [float or array of floats] Pressure(s) at given depth(s) in [Pa].

`radius`(*pressure*)

`v_p`(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_p [float or array of floats] P wave velocity at given depth(s) in [m/s].

v_phi(*depth*)

Parameters

depth_list [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_phi [float or array of floats] bulk sound wave velocity at given depth(s) in [m/s].

v_s(*depth*)

Parameters

depth [float or array of floats] Depth(s) [m] to evaluate seismic model at.

Returns

v_s [float or array of floats] S wave velocity at given depth(s) in [m/s].

5.12.4 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

v_p [float] corrected P wave velocity in [m/s].

v_s [float] corrected S wave velocitiy in [m/s].

v_phi [float] corrected Bulk sound velocity in [m/s].

5.13 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*
- *other*

5.13.1 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.Matas_etal_2007.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Matas_etal_2007.**al_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Matas_etal_2007.**ca_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Matas_etal_2007.**periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Matas_etal_2007.**wuestite**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.Matas_etal_2007.**ca_bridgmanite**
alias of *burnman.minerals.Matas_etal_2007.ca_perovskite*

burnman.minerals.Matas_etal_2007.**mg_bridgmanite**
alias of *burnman.minerals.Matas_etal_2007.mg_perovskite*

burnman.minerals.Matas_etal_2007.**fe_bridgmanite**
alias of *burnman.minerals.Matas_etal_2007.fe_perovskite*

burnman.minerals.Matas_etal_2007.**al_bridgmanite**
alias of *burnman.minerals.Matas_etal_2007.al_perovskite*

5.13.2 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_perovskite_3rdorder**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase**
Bases: *burnman.classes.mineral_helpers.HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_3rd**
Bases: *burnman.classes.mineral_helpers.HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS_3rd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS_3rd**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite**
alias of *burnman.minerals.Murakami_etal_2012.mg_perovskite*

burnman.minerals.Murakami_etal_2012.**fe_bridgmanite**
alias of *burnman.minerals.Murakami_etal_2012.fe_perovskite*

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite_3rdorder**
alias of *burnman.minerals.Murakami_etal_2012.mg_perovskite_3rdorder*

5.13.3 Murakami_2013

Minerals from Murakami 2013 and references therein.

class burnman.minerals.Murakami_2013.**periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_2013.**wuestite**
Bases: *burnman.classes.mineral.Mineral*

Murakami 2013 and references therein

class burnman.minerals.Murakami_2013.**mg_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.Murakami_2013.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.Murakami_2013.**mg_bridgmanite**
alias of *burnman.minerals.Murakami_2013.mg_perovskite*

burnman.minerals.Murakami_2013.**fe_bridgmanite**
alias of *burnman.minerals.Murakami_2013.fe_perovskite*

5.13.4 SLB_2005

Minerals from Stixrude & Lithgow-Bertelloni 2005 and references therein

class burnman.minerals.SLB_2005.**stishovite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2005.**periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2005.**wuestite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2005.**mg_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2005.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.SLB_2005.**mg_bridgmanite**
alias of *burnman.minerals.SLB_2005.mg_perovskite*

burnman.minerals.SLB_2005.**fe_bridgmanite**
alias of *burnman.minerals.SLB_2005.fe_perovskite*

5.13.5 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: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**ca_ferrite_structured_phase**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**clinopyroxene**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**garnet**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**akimotoite**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**ferropericlase**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**mg_fe_olivine**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**orthopyroxene**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**plagioclase**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**post_perovskite**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**mg_fe_perovskite**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**mg_fe_ringwoodite**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**mg_fe_aluminous_spinel**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**mg_fe_wadsleyite**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.SLB_2011.**anorthite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**albite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**spinel**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**hercynite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**forsterite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fayalite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_wadsleyite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_wadsleyite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_ringwoodite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_ringwoodite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**enstatite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**ferrosilite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_tschermaks**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**ortho_diopside**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**diopside**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**hedenbergite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**clinoenstatite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**ca_tschermaks**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**jadeite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**hp_clinoenstatite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**hp_clinoferrosilite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**ca_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_akimotoite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_akimotoite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**corundum**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**pyrope**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**almandine**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**grossular**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_majorite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**jd_majorite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**quartz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**coesite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**stishovite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**seifertite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**al_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_post_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_post_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**al_post_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**wuestite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**mg_ca_ferrite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**fe_ca_ferrite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**na_ca_ferrite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**kyanite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011.**nepheline**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.SLB_2011.**ab**
alias of *burnman.minerals.SLB_2011.albite*

burnman.minerals.SLB_2011.**an**
alias of *burnman.minerals.SLB_2011.anorthite*

burnman.minerals.SLB_2011.**sp**
alias of *burnman.minerals.SLB_2011.spinel*

burnman.minerals.SLB_2011.**hc**
alias of *burnman.minerals.SLB_2011.hercynite*

burnman.minerals.SLB_2011.**fo**
alias of *burnman.minerals.SLB_2011.forsterite*

burnman.minerals.SLB_2011.**fa**
alias of *burnman.minerals.SLB_2011.fayalite*

burnman.minerals.SLB_2011.**mgwa**
alias of *burnman.minerals.SLB_2011.mg_wadsleyite*

burnman.minerals.SLB_2011.**fewa**
alias of *burnman.minerals.SLB_2011.fe_wadsleyite*

burnman.minerals.SLB_2011.**mgri**
alias of *burnman.minerals.SLB_2011.mg_ringwoodite*

burnman.minerals.SLB_2011.**feri**
alias of *burnman.minerals.SLB_2011.fe_ringwoodite*

burnman.minerals.SLB_2011.**en**
alias of *burnman.minerals.SLB_2011.enstatite*

burnman.minerals.SLB_2011.**fs**
alias of *burnman.minerals.SLB_2011.ferrosilite*

burnman.minerals.SLB_2011.**mgts**
alias of *burnman.minerals.SLB_2011.mg_tschermaks*

burnman.minerals.SLB_2011.**odi**
alias of *burnman.minerals.SLB_2011.ortho_diopside*

burnman.minerals.SLB_2011.**di**
alias of *burnman.minerals.SLB_2011.diopside*

burnman.minerals.SLB_2011.**he**
alias of *burnman.minerals.SLB_2011.hedenbergite*

burnman.minerals.SLB_2011.**cen**
alias of *burnman.minerals.SLB_2011.clinoenstatite*

burnman.minerals.SLB_2011.**cats**
alias of *burnman.minerals.SLB_2011.ca_tschermaks*

burnman.minerals.SLB_2011.**jd**
alias of *burnman.minerals.SLB_2011.jadeite*

burnman.minerals.SLB_2011.**mgc2**
alias of *burnman.minerals.SLB_2011.hp_clinoenstatite*

burnman.minerals.SLB_2011.**fec2**
alias of *burnman.minerals.SLB_2011.hp_clinoferrosilite*

burnman.minerals.SLB_2011.**hpcen**
alias of *burnman.minerals.SLB_2011.hp_clinoenstatite*

burnman.minerals.SLB_2011.**hpcfs**
alias of *burnman.minerals.SLB_2011.hp_clinoferrosilite*

burnman.minerals.SLB_2011.**mgpv**
alias of *burnman.minerals.SLB_2011.mg_perovskite*

burnman.minerals.SLB_2011.**mg_bridgmanite**
alias of *burnman.minerals.SLB_2011.mg_perovskite*

burnman.minerals.SLB_2011.**fepv**
alias of *burnman.minerals.SLB_2011.fe_perovskite*

burnman.minerals.SLB_2011.**fe_bridgmanite**
alias of *burnman.minerals.SLB_2011.fe_perovskite*

burnman.minerals.SLB_2011.**alpv**
alias of *burnman.minerals.SLB_2011.al_perovskite*

burnman.minerals.SLB_2011.**capv**
alias of *burnman.minerals.SLB_2011.ca_perovskite*

burnman.minerals.SLB_2011.**mgil**
alias of *burnman.minerals.SLB_2011.mg_akimotoite*

burnman.minerals.SLB_2011.**feil**
alias of *burnman.minerals.SLB_2011.fe_akimotoite*

burnman.minerals.SLB_2011.**co**
alias of *burnman.minerals.SLB_2011.corundum*

burnman.minerals.SLB_2011.**py**
alias of *burnman.minerals.SLB_2011.pyrope*

burnman.minerals.SLB_2011.**al**
alias of *burnman.minerals.SLB_2011.almandine*

burnman.minerals.SLB_2011.**gr**
alias of *burnman.minerals.SLB_2011.grossular*

burnman.minerals.SLB_2011.**mgmj**
alias of *burnman.minerals.SLB_2011.mg_majorite*

burnman.minerals.SLB_2011.**jdmj**
alias of *burnman.minerals.SLB_2011.jd_majorite*

burnman.minerals.SLB_2011.**qtz**
alias of *burnman.minerals.SLB_2011.quartz*

burnman.minerals.SLB_2011.**coes**
alias of *burnman.minerals.SLB_2011.coesite*

burnman.minerals.SLB_2011.**st**
alias of *burnman.minerals.SLB_2011.stishovite*

burnman.minerals.SLB_2011.**seif**
alias of *burnman.minerals.SLB_2011.seifertite*

burnman.minerals.SLB_2011.**mppv**
alias of *burnman.minerals.SLB_2011.mg_post_perovskite*

burnman.minerals.SLB_2011.**fppv**
alias of *burnman.minerals.SLB_2011.fe_post_perovskite*

burnman.minerals.SLB_2011.**appv**
alias of *burnman.minerals.SLB_2011.al_post_perovskite*

burnman.minerals.SLB_2011.**pe**
alias of *burnman.minerals.SLB_2011.periclase*

burnman.minerals.SLB_2011.**wu**
alias of *burnman.minerals.SLB_2011.wuestite*

burnman.minerals.SLB_2011.**mgcf**
alias of *burnman.minerals.SLB_2011.mg_ca_ferrite*

burnman.minerals.SLB_2011.**fecf**
alias of *burnman.minerals.SLB_2011.fe_ca_ferrite*

burnman.minerals.SLB_2011.**nacf**
alias of *burnman.minerals.SLB_2011.na_ca_ferrite*

burnman.minerals.SLB_2011.**ky**
alias of *burnman.minerals.SLB_2011.kyanite*

burnman.minerals.SLB_2011.**neph**
alias of *burnman.minerals.SLB_2011.nepheline*

burnman.minerals.SLB_2011.**c2c**
alias of *burnman.minerals.SLB_2011.c2c_pyroxene*

burnman.minerals.SLB_2011.**cf**
alias of *burnman.minerals.SLB_2011.ca_ferrite_structured_phase*

burnman.minerals.SLB_2011.**cpx**
alias of *burnman.minerals.SLB_2011.clinopyroxene*

burnman.minerals.SLB_2011.**gt**
alias of *burnman.minerals.SLB_2011.garnet*

burnman.minerals.SLB_2011.**il**
alias of *burnman.minerals.SLB_2011.akimotoite*

burnman.minerals.SLB_2011.**ilmenite_group**
alias of *burnman.minerals.SLB_2011.akimotoite*

burnman.minerals.SLB_2011.**mw**
alias of *burnman.minerals.SLB_2011.ferropericlase*

burnman.minerals.SLB_2011.**magnesiowuestite**
alias of *burnman.minerals.SLB_2011.ferropericlase*

burnman.minerals.SLB_2011.**ol**
alias of *burnman.minerals.SLB_2011.mg_fe_olivine*

burnman.minerals.SLB_2011.**opx**
alias of *burnman.minerals.SLB_2011.orthopyroxene*

burnman.minerals.SLB_2011.**plag**
alias of *burnman.minerals.SLB_2011.plagioclase*

burnman.minerals.SLB_2011.**ppv**
alias of *burnman.minerals.SLB_2011.post_perovskite*

burnman.minerals.SLB_2011.**pv**
alias of *burnman.minerals.SLB_2011.mg_fe_perovskite*

burnman.minerals.SLB_2011.**mg_fe_bridgmanite**
alias of *burnman.minerals.SLB_2011.mg_fe_perovskite*

burnman.minerals.SLB_2011.**mg_fe_silicate_perovskite**
alias of *burnman.minerals.SLB_2011.mg_fe_perovskite*

burnman.minerals.SLB_2011.**ri**
alias of *burnman.minerals.SLB_2011.mg_fe_ringwoodite*

burnman.minerals.SLB_2011.**spinel_group**
alias of *burnman.minerals.SLB_2011.mg_fe_aluminous_spinel*

burnman.minerals.SLB_2011.**wa**
alias of *burnman.minerals.SLB_2011.mg_fe_wadsleyite*

burnman.minerals.SLB_2011.**spinelloid_III**
alias of *burnman.minerals.SLB_2011.mg_fe_wadsleyite*

5.13.6 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**periclase**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**wuestite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**mg_perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**fe_perovskite**
Bases: *burnman.classes.mineral.Mineral*

burnman.minerals.SLB_2011_ZSB_2013.**mg_bridgmanite**
alias of *burnman.minerals.SLB_2011_ZSB_2013.mg_perovskite*

burnman.minerals.SLB_2011_ZSB_2013.**fe_bridgmanite**
alias of *burnman.minerals.SLB_2011_ZSB_2013.fe_perovskite*

5.13.7 DKS_2013_solids

Solids from de Koker and Stixrude (2013) FPMD simulations

class burnman.minerals.DKS_2013_solids.**stishovite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_solids.**perovskite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_solids.**periclase**
Bases: *burnman.classes.mineral.Mineral*

5.13.8 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSiO3_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi2O5_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi3O7_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi5O11_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg2SiO4_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg3Si2O7_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg5SiO7_liquid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.DKS_2013_liquids.**MgO_liquid**
Bases: *burnman.classes.mineral.Mineral*

5.13.9 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: *burnman.classes.mineral.Mineral*

5.13.10 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**teph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lrn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mont**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**chum**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**chdr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mrw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**frw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**apv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**maj**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**py**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**alm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**spss**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**gr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**andr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**knor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**osma**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**osmm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**osfa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**vsv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**andalusite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ky**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sill**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**smul**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**amul**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**tpz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mnst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mnctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**merw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**spu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**zo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ep**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fep**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pmt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**law**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mpm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fpm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**jgd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**geh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**rnk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ty**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**crd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hcrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fcrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mncrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**phA**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cstn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**zrc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**en**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pren**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fs**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mgts**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**di**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hed**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**jd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**acm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**kos**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cats**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**caes**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**rhod**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pxmn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**wo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pswo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**wal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**tr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fact**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ts**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**parg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**gl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fgl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**rieb**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**anth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fanth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cumm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**grun**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ged**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**spr4**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**spr5**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fspr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mcar**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fcar**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**deer**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cel**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fcel**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ma**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**phl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ann**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mnbi**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**east**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**naph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**clin**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ames**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**afchl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**daph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mnchl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sud**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fsud**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**prl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ta**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fta**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**tats**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**tap**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**minn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**minm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**kao**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pre**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fpre**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**chr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**liz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**glt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fstp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mstp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**atg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ab**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**abh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mic**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**san**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**an**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**kcm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**wa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hol**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**q**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**trd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**crst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**coe**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**stv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ne**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cgh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sdl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**kls**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**me**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**wrk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lmt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**heu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**stlb**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**anl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lime**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ru**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**per**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**fper**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mang**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mcor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hem**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**esk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**bix**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**NiO**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pnt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**geik**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ilm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**bdy**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ten**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cup**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**herc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mft**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**usp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**picr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**br**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**dsp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**gth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**cc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**arag**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**mag**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**sid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**rhc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**dol**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**ank**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**syv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hlt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**pyr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**trot**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**tro**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lot**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**trov**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**any**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**iron**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**Ni**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**Cu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**gph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**diam**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**S**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**syvL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**hltL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**perL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**limL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**corL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**qL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**h2oL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**foL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**faL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**woL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**enL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**diL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**silL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**anL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**kspL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**abL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**neL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_ds62.**lcL**
Bases: *burnman.classes.mineral.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

5.13.11 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_fluids.**CH4**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_fluids.**O2**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_fluids.**H2**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_fluids.**S2**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HP_2011_fluids.**H2S**
Bases: *burnman.classes.mineral.Mineral*

5.13.12 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: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mrw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**frw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**apv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**npv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**cpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**maj**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**nagt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**py**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**alm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**gr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**en**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**cen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**hen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**hfs**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fs**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mgts**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**di**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**hed**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**jd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**cats**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**stv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**macf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mscf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fscf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**nacf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**cacf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**manal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**nanal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**msnal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fsnal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**canal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**per**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**fper**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**cor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HHPH_2013.**mcor**
Bases: *burnman.classes.mineral.Mineral*

5.13.13 HGP_2018_ds633

Endmember minerals 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. File autogenerated using HGP633data_to_burnman.py.

class burnman.minerals.HGP_2018_ds633.**fo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**teph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lrn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mont**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**chum**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**chdr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fwd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mrw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**frw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**apv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**npv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ppv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cpv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**maj**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**nagt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**py**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**alm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**spss**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**gr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**andr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ski**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**knor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**uv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**osma**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**osmm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**osfa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**vsv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**andalusite**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ky**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sill**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**smul**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**amul**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tpz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mnst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mnctd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**merw**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**spu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**zo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ep**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fep**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pmt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**law**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mpm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fpm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**jgd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**geh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ak**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**rnk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ty**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**crd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hcrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fcrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mncrd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**phA**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**phD**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**phE**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**shB**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cstn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**zrc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**zrt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tcn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**en**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pren**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hen**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hfs**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fs**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mgts**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**di**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hed**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**jd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kjd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**acm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kos**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cats**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**caes**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**rhod**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pxmn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**wo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pswo**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**wal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fact**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ts**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**parg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**gl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fgl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**nyb**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**rieb**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**anth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fanth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cumm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**grun**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ged**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**spr4**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**spr5**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fspr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mcar**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fcar**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**deer**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cel**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fcel**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ma**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**phl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ann**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mnbi**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**east**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**naph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tan**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**clin**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ames**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**afchl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**daph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mnchl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sud**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fsud**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**prl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ta**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fta**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tats**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tap**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**nta**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**minn**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**minm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kao**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pre**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fpre**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**chr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**liz**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**glt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fstp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mstp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**atg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ab**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**abh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mic**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**san**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**an**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kcm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**wa**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hol**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**q**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**trd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**crst**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**coe**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**stv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ne**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cg**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cgh**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**macf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mscf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fscf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**nacf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cacf**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**manal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**nanal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**msnal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fsnal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**canal**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sdl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kls**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**me**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**wrk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lmt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**heu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**stlb**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**anl**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lime**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ru**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**per**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**fper**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**wu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mang**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mcor**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hem**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**esk**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**bix**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**NiO**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pnt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**geik**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ilm**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**bdy**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyT**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyC**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ten**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cup**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**herc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mft**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**qnd**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**usp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**picr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**br**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**dsp**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**gth**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**cc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**arag**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**mag**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**sid**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**rhc**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**dol**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ank**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**syv**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hlt**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**pyr**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**trot**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**tro**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lot**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**trov**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**any**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**iron**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**Ni**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**Cu**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**gph**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**diam**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**S**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**syvL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hltL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**perL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**limL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**corL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**eskL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**hemL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**qL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**h2oL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**foL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**faL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**woL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**enL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**diL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**silL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**anL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**kspL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**abL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**neL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**lcL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**ruL**
Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyL**
Bases: *burnman.classes.mineral.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

5.13.14 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: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**plagioclase**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**clinopyroxene**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**cfs**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**crdi**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**cess**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**cen**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**cfm**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**olivine**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**spinel**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**garnet**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**orthopyroxene**(*molar_fractions=None*)
Bases: *burnman.classes.solidsolution.SolidSolution*

class burnman.minerals.JH_2015.**fm**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**odi**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**cren**
Bases: burnman.classes.combinedmineral.CombinedMineral

class burnman.minerals.JH_2015.**mess**
Bases: burnman.classes.combinedmineral.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

5.13.15 Other minerals

class burnman.minerals.other.**liquid_iron**

Bases: *burnman.classes.mineral.Mineral*

Liquid iron equation of state from Anderson and Ahrens (1994)

class burnman.minerals.other.**ZSB_2013_mg_perovskite**

Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.other.**ZSB_2013_fe_perovskite**

Bases: *burnman.classes.mineral.Mineral*

class burnman.minerals.other.**Speziale_fe_periclase**

Bases: *burnman.classes.mineral_helpers.HelperSpinTransition*

class burnman.minerals.other.**Speziale_fe_periclase_HS**

Bases: *burnman.classes.mineral.Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Speziale_fe_periclase_LS**

Bases: *burnman.classes.mineral.Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Liquid_Fe_Anderson**

Bases: *burnman.classes.mineral.Mineral*

Anderson & Ahrens, 1994 JGR

class burnman.minerals.other.**Fe_Dewaele**
Bases: *burnman.classes.mineral.Mineral*

Dewaele et al., 2006, Physical Review Letters

5.14 Tools

Burnman has a number of high-level tools to help achieve common goals. Several of these have already been described in previous sections.

5.14.1 Mathematical

burnman.tools.math.**round_to_n**(*x*, *xerr*, *n*)

burnman.tools.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.tools.math.**float_eq**(*a*, *b*)
Test if two floats are almost equal to each other

burnman.tools.math.**linear_interpol**(*x*, *x1*, *x2*, *y1*, *y2*)
Linearly interpolate to point x, between the points (x1,y1), (x2,y2)

burnman.tools.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 [parameter list] Additional arguments to give to fn

ratio : 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: floats 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.

burnman.tools.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 (default=4.)] The number of standard deviations at which to truncate the smoothing.

mode [{'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

smoothed_array: numpy ndarray The smoothed array

`burnman.tools.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.interp2d() interpolators which can be used to query the array, or its derivatives in the x- and y- directions.

Parameters

array [2D numpy array] The array to smooth. Each element array[i][j] corresponds to the position x_values[i], y_values[j]

x_values [1D numpy array] The gridded x values over which to create the smoothed grid

y_values [1D numpy array] 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 x axis

truncate [float (optional)] The number of standard deviations at which to truncate the smoothing (default = 4.).

mode [{'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 [{'xy', 'ij'}, optional] Cartesian ('xy', default) or matrix ('ij') indexing of output. See numpy.meshgrid for more details.

Returns

interps: tuple of three interp2d functors interpolation functions for the smoothed property and the first derivatives with respect to x and y.

burnman.tools.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.tools.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.tools.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.tools.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.tools.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.tools.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

indices [1D numpy array of integers] The indices corresponding to a set of independent rows of the input array.

burnman.tools.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

complete_basis [2D numpy array] An array containing the basis vectors spanning both of the input arrays.

5.14.2 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*)

Plot types must be one of: '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 objects must be initialised with projection='polar'

5.14.3 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 [string] 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 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 [string] Name of model, appears in name of output file

axisem_ref [string] Reference file, used to copy the header and for the rest of the planet, in the case of a Layer(), default = 'axisem_prem_ani_noocean.txt'

plotting: Boolean True means plot of the old model and replaced model will be shown (default = False)

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 axisem file

modelname [string] Name of model, appears in name of output file

mineos_ref [string] Reference file, used to copy the header and for the rest of the planet, in the case of a Layer(), default = 'mineos_prem_noocean.txt'

plotting: Boolean True means plot of the old model and replaced model will be shown (default = False)

5.14.4 Miscellaneous

class burnman.tools.misc.**OrderedCounter**(***kwds*)

Bases: collections.Counter, collections.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 >>> prime_factors = Counter({2: 2, 3: 3, 17: 1}) >>> product = 1 >>> for factor in prime_factors.elements(): # loop over factors ... product *= factor # and multiply them >>> product 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*)

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('abcdeabcdabcaba').most_common(3)
[('a', 5), ('b', 4), ('c', 3)]
```

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(*k*[, *d*]) → v, remove specified key and return the corresponding
value. If key is not found, d is returned if given, otherwise KeyError is raised.

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(***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
```

update(***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.tools.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.tools.misc.**merge_two_dicts**(*x*, *y*)

Given two dicts, merge them into a new dict as a shallow copy.

burnman.tools.misc.**flatten**(*arr*)

burnman.tools.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.tools.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.tools.misc.**sort_table**(*table*, *col=0*)

Sort the table according to the column number

burnman.tools.misc.**read_table**(*filename*)

burnman.tools.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.tools.misc.**cut_table**(*table*, *min_value*, *max_value*)

burnman.tools.misc.**lookup_and_interpolate**(*table_x*, *table_y*, *x_value*)

burnman.tools.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 [Material] The material instance evaluated by the output function.

attributes [list of strings] 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

——-

f [function(x)] Function which returns the value of product(a_i**p_i) as a function of condition (x = [P, T, V])

5.14.5 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*)

Compute numerical derivatives of the gibbs free energy of a mineral under given conditions, and check these values against those provided analytically by the equation of state

Parameters

m [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 [boolean] Decide whether to print information about each check

including_shear_properties [boolean] Decide whether to check shear information, which is pointless for liquids and equations of state without shear modulus parameterizations

Returns

consistency: boolean If all checks pass, returns True

burnman.tools.eos.**check_anisotropic_eos_consistency**(*m*, *P=1000000000.0*, *T=2000.0*, *tol=0.0001*, *verbose=False*)

Compute numerical derivatives of the gibbs free energy of a mineral under given conditions, and check these values against those provided analytically by the equation of state

Parameters

m [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 [boolean] Decide whether to print information about each check

Returns

consistency: boolean If all checks pass, returns True

5.14.6 Unit cell

burnman.tools.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

V [float] Volume [m^3/mol]

burnman.tools.unitcell.**cell_parameters_to_vectors**(*cell_parameters*)

Converts cell parameters to unit cell vectors.

Parameters

cell_parameters [1D numpy array] 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

M [2D numpy array] 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.

burnman.tools.unitcell.**cell_vectors_to_parameters**(*M*)

Converts unit cell vectors to cell parameters.

Parameters

M [2D numpy array] 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

cell_parameters [1D numpy array] 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.

CHAPTER

SIX

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 [Myh21]. Examples are provided in the file examples/example_anisotropic_mineral.py.

  Bob Myhill, 2021/10/03

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.

[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 Dziewoński. 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.

[Myh21] R. Myhill. An anisotropic equation of state for high pressure, high temperature applications. *EarthArXiv*, pages 1–24, 2021. doi:10.31223/X5561D.

[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.

[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.

[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.

[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.

[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.

[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

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z