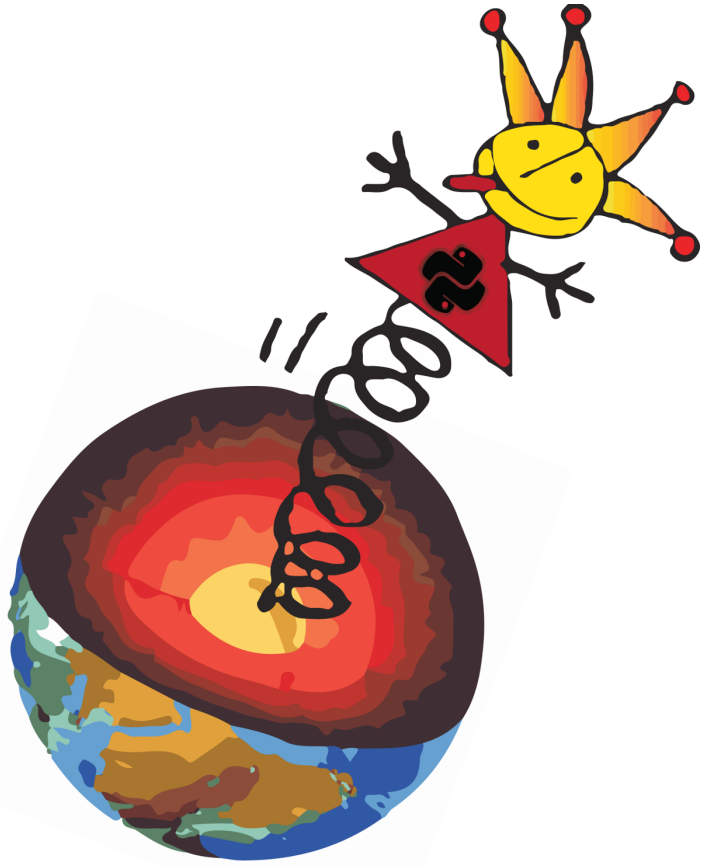
COMPUTATIONAL INFRASTRUCTURE FOR GEODYNAMICS (CIG)

BurnMan

a thermodynamics and thermoelasticity toolkit

User Manual

Version 3.0.0a0



Robert Myhill
Sanne Cottaar
Timo Heister
Ian Rose
Cayman Unterborn
Benedict Heinen
Robert Farla
Juliane Dannberg
Rene Gassmoeller

http://geodynamics.org

CONTENTS

1 Introducing BurnMan 3.0 **1**
- 1.1 Overview . . . 1
- 1.2 Structure . . . 2
- 1.3 Requirements . . . 3
  - 1.3.1 Optional modules . . . 3
- 1.4 Installation . . . 3
  - 1.4.1 Dependencies . . . 3
  - 1.4.2 Stable version . . . 3
  - 1.4.3 Development version . . . 4
  - 1.4.4 Checking that the installation worked . . . 4
- 1.5 Citing BurnMan . . . 4
- 1.6 Contributing to BurnMan . . . 4
- 1.7 Acknowledgement and Support . . . 5

2 The BurnMan Tutorials **7**
- 2.1 Part 1: Material Classes . . . 7
  - 2.1.1 Introduction . . . 7
  - 2.1.2 Acknowledgments . . . 7
  - 2.1.3 Getting started with BurnMan . . . 8
  - 2.1.4 Types of BurnMan Material objects . . . 8
  - 2.1.5 Mineral objects . . . 8
    - 2.1.5.1 Equations of state . . . 8
    - 2.1.5.2 Property modifiers . . . 10
    - 2.1.5.3 The CombinedMineral class . . . 11
    - 2.1.5.4 Implemented datasets . . . 11
    - 2.1.5.5 Interrogating the properties of a Mineral object at a given pressure and temperature . . . 12
  - 2.1.6 Solution objects . . . 14
    - 2.1.6.1 Model formalisms . . . 15
    - 2.1.6.2 Implemented datasets . . . 18
    - 2.1.6.3 Interrogating the properties of a Solution object at a given composition, pressure and temperature . . . 18
  - 2.1.7 Composite objects . . . 21
- 2.2 Part 2: The Composition Class . . . 23
  - 2.2.1 Introduction . . . 23

2.2.2 The Composition class . . . 23
2.3 Part 3: Layers and Planets . . . 25
2.3.1 Introduction . . . 26
2.3.2 Building a planet . . . 26
2.3.3 The Layer class . . . 26
2.3.4 The Planet class . . . 28
2.4 Part 4: Fitting . . . 36
2.4.1 Introduction . . . 36
2.4.2 Fitting parameters for an equation of state to experimental data . . . 37
2.4.3 Finding the best fit endmember proportions of a solution given a bulk composition 43
2.4.4 Fitting phase proportions to a bulk composition . . . 46
2.5 Part 5: Equilibrium problems . . . 50
2.5.1 Introduction . . . 50
2.5.2 Phase equilibria . . . 50
2.5.2.1 What BurnMan does and doesn't do . . . 50
2.5.3 Equilibrating at fixed bulk composition . . . 51
2.5.4 Equilibrating while allowing bulk composition to vary . . . 54
2.6 CIDER Tutorial 2014 . . . 57
2.6.1 CIDER 2014 BurnMan Tutorial — step 1 . . . 57
2.6.2 CIDER 2014 BurnMan Tutorial — step 2 . . . 58
2.6.3 CIDER 2014 BurnMan Tutorial — step 3 . . . 59

3 Materials **61**
3.1 Material Base Class . . . 61
3.1.1 Overview . . . 61
3.1.2 Property Evaluation . . . 61
3.1.3 State Management . . . 62
3.1.4 Introspection and Debugging . . . 63
3.1.5 Evaluation Routines . . . 63
3.2 Mineral Classes . . . 63
3.2.1 Mineral Base Class . . . 63
3.2.1.1 Key Features . . . 64
3.2.1.2 Mineral Datasets . . . 64
3.2.1.3 Instantiation . . . 64
3.2.2 Combined Mineral Class . . . 64
3.2.3 Equations of State for Minerals . . . 65
3.2.3.1 Isothermal Equations of State . . . 66
3.2.3.2 Thermal Equations of State . . . 72
3.2.4 Mineral Property Modifiers . . . 86
3.2.4.1 Landau tricritical model (Putnis, 1992) (available as landau) . . . 88
3.2.4.2 Landau tricritical model (Stixrude and Lithgow-Bertelloni, 2022) (available as landau_slb_2022) . . . 89
3.2.4.3 Landau tricritical model (Holland and Powell, 1998) (available as landau_hp) 90
3.2.4.4 Linear in P and T (available as linear) . . . 91
3.2.4.5 Bragg-Williams model (available as bragg_williams) . . . 91
3.2.4.6 Magnetic ordering (Chin et al., 1987) (available as magnetic_chs) . . . 92
3.2.4.7 Debye model excess (Helmholtz) (available as debye) . . . 92
3.2.4.8 Debye model excess (internal energy) (available as debye_delta) . . . 93

3.2.4.9 Einstein model excess (Helmholtz) (available as einstein) . . . 93
3.2.4.10 Einstein model excess (internal energy) (available as einstein_delta) . . . 94
3.3 Solutions . . . 95
3.3.1 The Solution Class . . . 95
3.3.2 Solution Model Classes . . . 96
3.3.2.1 Mean-field solution models . . . 96
3.3.2.2 Implemented solution models . . . 97
3.4 Elastic Solution Class . . . 101
3.4.1 The Elastic Solution Class . . . 102
3.4.2 Elastic Solution Model Classes . . . 102
3.4.2.1 Mean-field elastic solution models . . . 102
3.4.2.2 Implemented elastic solution models . . . 102
3.5 Composite Class . . . 105
3.6 PerpleX Class . . . 106

4 Anisotropic Materials **109**
4.1 Anisotropic Materials . . . 109
4.1.1 Anisotropic Material Base Class . . . 109
4.1.2 Derived classes . . . 112
4.2 Anisotropic Mineral Class . . . 113
4.3 Anisotropic Solution Class . . . 114
4.4 Hyperelastic Mineral Class . . . 114
4.4.1 Cell geometry and the hydrostatic reference . . . 115
4.4.2 Helmholtz equation of state . . . 115
4.4.2.1 Why the hydrostatic spatial stiffness appears . . . 115
4.4.3 Stress, entropy and numerical derivatives . . . 116
4.4.4 Elastic stiffnesses . . . 117
4.4.4.1 Relation to material elasticity . . . 117
4.4.4.2 Seismic wave propagation . . . 117
4.4.5 Maxwell relations and thermal response . . . 118
4.4.6 Rotations and output frames . . . 119
4.4.7 Using the model . . . 119
4.4.8 Class reference . . . 120

5 Relaxed Material Classes **121**
5.1 Overview . . . 121
5.2 Mathematical Derivation . . . 121
5.3 Relaxed Material Classes . . . 123
5.3.1 Relaxed Solution Class . . . 123
5.3.2 Relaxed Anisotropic Solution Class . . . 123
5.3.3 Relaxed Composite Class . . . 124

6 Averaging Schemes **127**
6.1 Overview . . . 127
6.2 Implemented Averaging Schemes . . . 128
6.2.1 VoigtReussHill . . . 128
6.2.2 Voigt . . . 128
6.2.3 Reuss . . . 128

6.2.4 HashinShtrikmanUpper . . . 129
6.2.5 HashinShtrikmanLower . . . 129
6.2.6 HashinShtrikmanAverage . . . 129
6.3 Computing seismic velocities . . . 129
6.4 Attenuation . . . 129

7 Mineral Datasets **131**

8 Calibrants **133**

9 Seismic Models **135**
9.1 Overview . . . 135
9.2 Comparison to Seismic profiles . . . 135

10 Geotherms **137**

11 Composition Class **139**

12 Layers and Planets **141**

13 Material Polytopes **143**

14 Fitting Model Parameters to Data **145**
14.1 Fitting Model Compositions to Data . . . 145
14.1.1 Overview . . . 145
14.1.2 Available Fitting Functions . . . 145
14.2 Fitting Equation of State Parameters to Data . . . 147
14.2.1 Overview . . . 147
14.2.2 Available Fitting Functions . . . 148
14.3 Fitting Solution Model Parameters to Data . . . 150
14.3.1 Overview . . . 150
14.3.2 Available Fitting Functions . . . 150

15 Equilibration **153**
15.1 Overview . . . 153
15.2 Implemented function . . . 153

16 Thermobarometry **157**
16.1 Overview . . . 157
16.2 Implemented function . . . 157

17 Tools **161**

18 Utility Functions **163**

19 Examples **165**
19.1 Class examples . . . 165
19.1.1 example_mineral . . . 165
19.1.2 example_gibbs_modifiers . . . 166
19.1.3 example_solution . . . 169

19.1.4 example_composite . . . 172
19.1.5 example_calibrants . . . 174
19.1.6 example_anisotropy . . . 174
19.1.7 example_anisotropic_mineral . . . 175
19.1.8 example_geotherms . . . 177
19.1.9 example_composition . . . 178
19.2 Simple Examples . . . 178
19.2.1 example_beginner . . . 179
19.2.2 example_seismic . . . 180
19.2.3 example_composite_seismic_velocities . . . 182
19.2.4 example_averaging . . . 183
19.2.5 example_chemical_potentials . . . 185
19.3 More Advanced Examples . . . 185
19.3.1 example_hyperelastic_stress_rates . . . 186
19.3.2 example_spintransition . . . 190
19.3.3 example_spintransition_thermal . . . 191
19.3.4 example_user_input_material . . . 192
19.3.5 example_optimize_pv . . . 192
19.3.6 example_compare_all_methods . . . 193
19.3.7 example_build_planet . . . 194
19.3.8 example_fit_composition . . . 197
19.3.9 example_fit_data . . . 198
19.3.10 example_fit_eos . . . 200
19.3.11 example_fit_solution . . . 207
19.3.12 example_equilibrate . . . 207
19.3.13 example_olivine_binary . . . 219
19.4 Reproducing Cottaar, Heister, Rose and Unterborn (2014) . . . 220
19.4.1 paper_averaging . . . 220
19.4.2 paper_benchmark . . . 220
19.4.3 paper_fit_data . . . 220
19.4.4 paper_incorrect_averaging . . . 220
19.4.5 paper_opt_pv . . . 220
19.4.6 paper_onefit . . . 221
19.4.7 paper_uncertain . . . 221
19.5 Misc or work in progress . . . 221
19.5.1 example_grid . . . 221
19.5.2 example_woutput . . . 221

20 Autogenerated Full API **223**
20.1 Materials . . . 223
20.1.1 Material Base Class . . . 223
20.2 Perple_X Class . . . 235
20.3 Minerals . . . 248
20.4 Solutions . . . 267
20.5 Mineral helpers . . . 288
20.6 Anisotropic materials . . . 297
20.7 Composites . . . 331
20.8 Calibrants . . . 352

20.9 Equation of State Classes and Functions . . . 353
20.9.1 Isothermal Equations of State . . . 355
20.9.1.1 Base class . . . 355
20.9.1.2 birch murnaghan . . . 359
20.9.1.3 macaw . . . 375
20.9.1.4 modified tait . . . 379
20.9.1.5 morse potential . . . 382
20.9.1.6 murnaghan . . . 386
20.9.1.7 reciprocal kprime . . . 390
20.9.1.8 spock . . . 393
20.9.1.9 vinet . . . 397
20.9.2 Thermal Equations of State . . . 401
20.9.2.1 Base class . . . 401
20.9.2.2 aa . . . 407
20.9.2.3 brosh calphad . . . 411
20.9.2.4 cork . . . 416
20.9.2.5 dks liquid . . . 420
20.9.2.6 dks solid . . . 424
20.9.2.7 hp . . . 427
20.9.2.8 mie grueneisen debye . . . 439
20.9.2.9 modular mie grueneisen debye . . . 450
20.9.2.10 slb . . . 460
20.9.3 Other Equation of State classes . . . 475
20.9.3.1 anharmonic debye . . . 475
20.9.3.2 anharmonic prefactor models . . . 476
20.9.3.3 anharmonic thermal models . . . 477
20.9.3.4 debye temperature models . . . 478
20.9.4 Equation of State Functions . . . 478
20.9.4.1 anharmonic thermal models . . . 478
20.9.4.2 birch murnaghan . . . 478
20.9.4.3 bukowinski electronic . . . 480
20.9.4.4 debye . . . 481
20.9.4.5 debye temperature models . . . 481
20.9.4.6 einstein . . . 482
20.9.4.7 macaw . . . 482
20.9.4.8 modified tait . . . 482
20.9.4.9 morse potential . . . 483
20.9.4.10 murnaghan . . . 483
20.9.4.11 property modifiers . . . 483
20.9.4.12 reciprocal kprime . . . 489
20.9.4.13 vinet . . . 490
20.10 Solution models . . . 490
20.10.1 Base classes . . . 491
20.10.2 Relaxed solutions . . . 534
20.10.3 Mechanical solution . . . 564
20.10.4 Ideal solution . . . 570
20.10.5 Asymmetric regular solution . . . 577
20.10.6 Symmetric regular solution . . . 584

20.10.7 Subregular solution . . . 591
20.10.8 Function solution . . . 599
20.11 Solution tools . . . 606
20.12 Compositions . . . 607
20.12.1 Base class . . . 607
20.12.2 Utility functions . . . 609
20.12.3 Fitting functions . . . 610
20.13 Polytopes . . . 612
20.13.1 Base class . . . 612
20.13.2 Polytope tools . . . 614
20.14 Averaging Schemes . . . 616
20.14.1 Base class . . . 616
20.14.2 Voigt bound . . . 617
20.14.3 Reuss bound . . . 618
20.14.4 Voigt-Reuss-Hill average . . . 619
20.14.5 Hashin-Shtrikman upper bound . . . 620
20.14.6 Hashin-Shtrikman lower bound . . . 621
20.14.7 Hashin-Shtrikman arithmetic average . . . 622
20.15 Geotherms . . . 623
20.16 Layers and Planets . . . 629
20.16.1 Layer . . . 629
20.16.2 Planet . . . 639
20.17 Thermodynamics . . . 646
20.17.1 Lattice Vibrations . . . 646
20.17.1.1 Debye model . . . 646
20.17.1.2 Einstein model . . . 647
20.17.2 Chemistry parsing and thermodynamics . . . 648
20.18 Equilibrium Thermodynamics . . . 656
20.19 Seismic Models . . . 659
20.19.1 Base class for all seismic models . . . 659
20.19.2 Class for 1D Models . . . 662
20.19.3 Models currently implemented . . . 665
20.19.4 Attenuation Correction . . . 685
20.20 Mineral databases . . . 685
20.20.1 DKS_2013_liquids . . . 686
20.20.2 DKS_2013_solids . . . 687
20.20.3 HGP_2018_ds633 . . . 687
20.20.4 HHPH_2013 . . . 703
20.20.5 HP_2011_ds62 . . . 705
20.20.6 HP_2011_fluids . . . 719
20.20.7 Irving et al. 2018 . . . 720
20.20.8 JH_2015 . . . 720
20.20.9 Matas_etal_2007 . . . 723
20.20.10Murakami_2013 . . . 723
20.20.11Murakami_etal_2012 . . . 724
20.20.12RS_2014_liquids . . . 725
20.20.13Saxena and Eriksson (2015) . . . 725
20.20.14SLB_2005 . . . 725

20.20.15SLB_2011 . . . 726
20.20.16SLB_2011_ZSB_2013 . . . 733
20.20.17SLB_2024 . . . 743
20.20.18ig50NCKFMASHTOCr . . . 754
20.20.19ig50NCKFMASTOCr . . . 755
20.20.20mb50NCKFMASHTO . . . 756
20.20.21mp50KFMASH . . . 758
20.20.22mp50MnNCKFMASHTO . . . 758
20.20.23mp50NCKFMASHTO . . . 760
20.20.24Other minerals . . . 761
20.21 Calibrant databases . . . 762
20.21.1 Anderson_1989 . . . 763
20.21.2 Armentrout_2015 . . . 763
20.21.3 Campbell_2009 . . . 763
20.21.4 Chidester_2021 . . . 763
20.21.5 Decker_1971 . . . 764
20.21.6 Dewaele_2008 . . . 764
20.21.7 Dewaele_2012 . . . 764
20.21.8 Dewaele_2013 . . . 764
20.21.9 Dewaele_2020 . . . 764
20.21.10Dorogokupets_2007 . . . 765
20.21.11Dorogokupets_2017 . . . 766
20.21.12Dubrovinsky_1998 . . . 766
20.21.13Fei_2007 . . . 766
20.21.14Fei_2016 . . . 766
20.21.15Holmes_1989 . . . 766
20.21.16Huang_2016 . . . 767
20.21.17LeGodec_2000 . . . 767
20.21.18Litasov_2013 . . . 767
20.21.19Matsui_2009 . . . 767
20.21.20Matsui_2010 . . . 767
20.21.21Matsui_2012 . . . 767
20.21.22Miozzi_2020 . . . 768
20.21.23Ono_2022 . . . 768
20.21.24Pigott_2015 . . . 768
20.21.25Shi_2022 . . . 768
20.21.26Shim_2002 . . . 768
20.21.27Sokolova_2013 . . . 768
20.21.28Speziale_2001 . . . 768
20.21.29Tange_2009 . . . 769
20.21.30Tateno_2019 . . . 769
20.21.31Tsuchiya_2003 . . . 769
20.21.32Walker_2002 . . . 769
20.21.33Zeng_2010 . . . 769
20.21.34Zha_2004 . . . 769
20.21.35Zha_2008 . . . 770
20.21.36Zhang_1999 . . . 770
20.21.37Zhao_1997 . . . 770

20.21.38Zhao_2000 . . . 770
20.22 Optimization . . . 771
20.23 Tools . . . 802
20.23.1 Unit cell . . . 802
20.23.2 Equilibration . . . 802
20.23.3 Optimal thermobarometry . . . 809
20.23.4 Equations of state . . . 813
20.23.5 Solution . . . 815
20.23.6 Chemistry . . . 816
20.23.7 Partitioning . . . 819
20.23.8 Plotting . . . 820
20.23.9 Seismic . . . 821
20.23.10Output for seismology . . . 821
20.23.11Polytopes . . . 823
20.24 Utilities . . . 826
20.24.1 Unit cell . . . 826
20.24.2 Chemistry . . . 827
20.24.3 Anisotropy . . . 833
20.24.4 Plotting . . . 834
20.24.5 Mathematical . . . 835
20.24.6 Row reductions and independent rows . . . 840
20.24.7 Miscellaneous . . . 840

21 Changes **847**

22 2021-08-02 **849**

23 2021-08-04 **851**

24 2021-08-05 **853**

25 2021-09-27 **855**

26 2021-10-03 **857**

27 2021-10-08 **859**

28 2021-10-22 **861**

29 2021-10-22 **863**

30 2021-10-22 **865**

31 2021-10-30 **867**

32 2021-11-02 **869**

33 2022-01-28 **871**

34 2022-02-05 **873**

35 2022-02-05 875

36 2022-02-07 877

37 2022-02-16 879

38 2022-02-16 881

39 2022-06-24 883

40 2022-06-25 885

41 2022-10-03 887

42 2022-10-22 889

43 2022-10-22 891

44 2022-10-23 893

45 2023-06-20 895

Bibliography 897

Index 907

CHAPTER

ONE

INTRODUCING BURNMAN 3.0

1.1 Overview

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

Properties which BurnMan can calculate include:

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

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

- the full codebase, which includes implementations of many static and thermal equations of state (including Vinet, Birch Murnaghan, Mie-Debye-Grueneisen, Modified Tait), and solution models (ideal, symmetric, asymmetric, subregular).
- popular endmember and solution datasets already coded into burnman-usable format (including [HollandPowell11], [SLB05], [SLB11] and [SLB22])
- Optimal least squares fitting routines for multivariate data with (potentially correlated) errors in pressure and temperature. As an example, such functions can be used to simultaneously fit volumes, seismic velocities and enthalpies.
- a "Planet" class, which self-consistently calculates gravity profiles, mass, moment of inertia of planets given the chemical and temperature structure of a planet
- published geotherms
- a tutorial on the basic use of BurnMan
- a large collection of annotated examples
- a set of high-level functions which create files readable by seismological and geodynamic software, including: Mineos [MWF11], AxiSEM [NissenMeyervanDrielStahler+14] and ASPECT

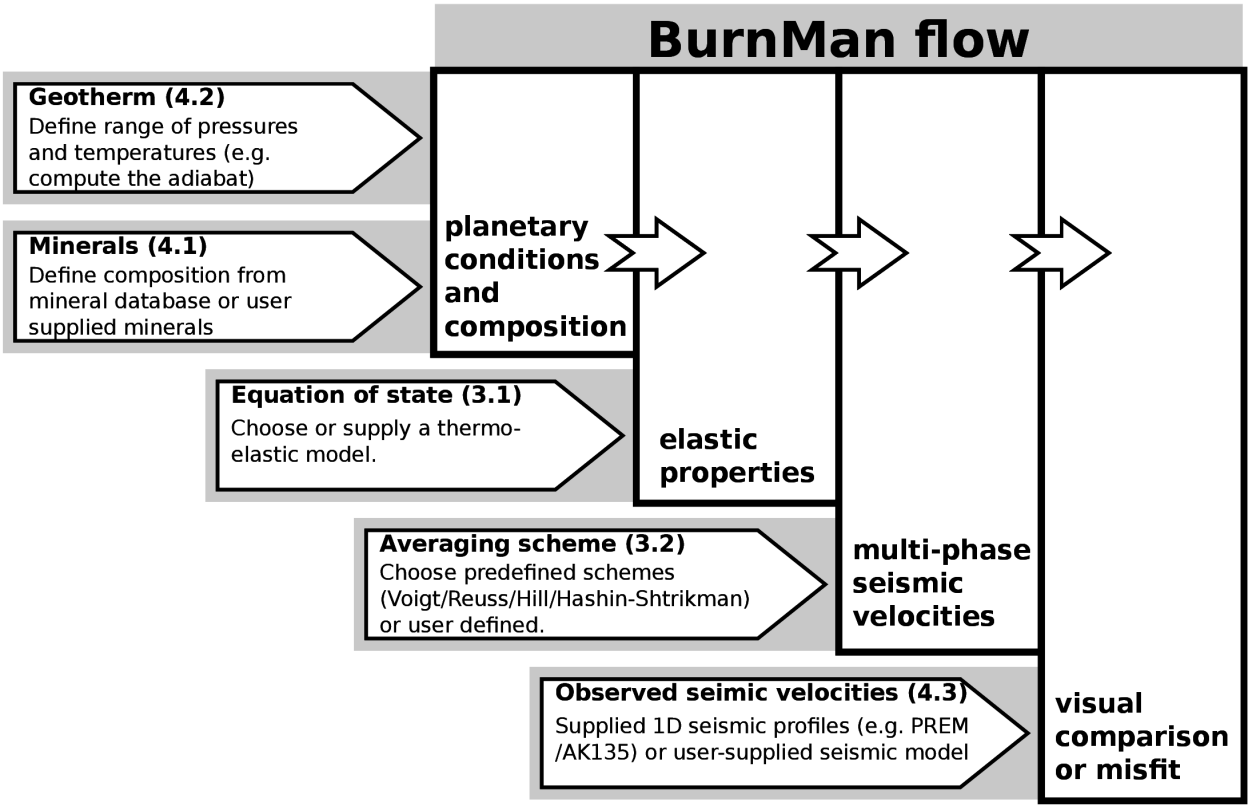
- an extensive suite of unit tests to ensure code functions as intended
- a series of benchmarks comparing BurnMan output with published data
- a directory containing user-contributed code from published papers

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

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

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

1.2 Structure



1.3 Requirements

- Python 3.10+
- Python modules: NumPy, SciPy, SymPy, Sparse, Matplotlib

1.3.1 Optional modules

Needed for some functionality:

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

1.4 Installation

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

1.4.1 Dependencies

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

    python –version

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

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

    python -m ensurepip –upgrade

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

1.4.2 Stable version

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

    python -m pip install burnman

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

1.4.3 Development version

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

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

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

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

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

1.4.4 Checking that the installation worked

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

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

    python example_beginner.py

If figures show up, BurnMan has been installed.

1.5 Citing BurnMan

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

- Myhill, R., Cottaar, S., Heister, T., Rose, I., Unterborn, C., Dannberg, J. and Gassmoeller, R. (2023). BurnMan - a Python toolkit for planetary geophysics, geochemistry and thermodynamics. Journal of Open Source Software. (https://doi.org/10.21105/joss.05389)
- Myhill, R., Cottaar, S., Heister, T., Rose, I., Unterborn, C., Dannberg, J., Gassmoeller, R. and Farla, R. (2024): BurnMan v2.1.0 [Software]. Computational Infrastructure for Geodynamics. Zenodo. (https://doi.org/10.5281/zenodo.14238360)
- Cottaar S., Heister, T., Rose, I., and Unterborn, C., (2014). BurnMan: A lower mantle mineral physics toolkit, Geochemistry, Geophysics, and Geosystems, 15(4), 1164-1179 (https://doi.org/10.1002/2013GC005122)

1.6 Contributing to BurnMan

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

BurnMan also includes a contrib directory that contains python and ipython scripts used to reproduce published results. We welcome the submission of new contributions to this directory. As with the contribution

of code, please contact us at info@burnman.org or make a pull request at https://github.com/geodynamics/burnman.

1.7 Acknowledgement and Support

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

CHAPTER

TWO

THE BURNMAN TUTORIALS

These tutorials introduce users to the main classes and some of the important functions implemented in BurnMan. The tutorials are split according to topic, and each tutorial is implemented as a Jupyter notebook.

An older tutorial from 2014 is also included.

The BurnMan Tutorial

2.1 Part 1: Material Classes

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

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

2.1.1 Introduction

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

Demonstrates

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

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

2.1.2 Acknowledgments

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

This project was initiated at, and follow-up research support was received through, CIDER (NSF FESD grant 1135452). The development of BurnMan has been supported by the Computational Infrastructure for Geodynamics initiative (CIG), through the Science and Technologies Funding Council (U.K.) under Award

No. ST/R001332/1 and through the Natural Environment Research Council (U.K.) Large Grant MC-squared (Award No. NE/T012633/1). The authors have also received support from the University of California - Davis.

2.1.3 Getting started with BurnMan

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

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

2.1.4 Types of BurnMan Material objects

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

2.1.5 Mineral objects

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

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

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

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

2.1.5.1 Equations of state

BurnMan identifies the desired equation of state by checking the value of the string parameter "equation_of_state" in the parameters dictionary that is passed as an argument to the Mineral class. BurnMan currently contains implementations of the following static equations of state (i.e., equations of state with no temperature dependence):

- "bm2", "bm3" and "bm4": Birch-Murnaghan (2nd, 3rd and 4th order)
- "mt": Modified Tait
- "morse": Morse potential
- "vinet": Vinet (originally the Rydberg equation of state)

- "rkprime": Reciprocal K-prime

And also the following thermal equations of state:

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```
                      'eta_s_0': 2.29972,
                      'n': sum(forsterite_formula.values()),
                      'molar_mass': molar_mass}

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

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

2.1.5.2 Property modifiers

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

BurnMan allows users to apply modifications to the Gibbs energy $\mathcal{G}(P,T)$ via a "property_modifiers" attribute. This attribute takes the form of a list of different modifiers, which are themselves each composed of a list consisting of an identifying string and a dictionary containing the required parameters. The following modifiers are currently implemented in BurnMan:

- "linear": Linear in pressure and temperature (i.e. $\Delta\mathcal{G} = \Delta\mathcal{E} - T\Delta S + P\Delta V$)
- "landau": A Landau order-disorder transition (Putnis, 1992)
- "landau_hp": A Landau order-disorder transition (Holland and Powell, 2011)
- "bragg_williams": A Bragg-Williams order-disorder transition
- "magnetic_chs": A magnetic order-disorder transition

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

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

(continues on next page)

(continued from previous page)

```
                  'molar_mass': 0.1596882}

# Dictionary for Landau transition which modifieds the Gibbs energy
hematite_property_modifiers = [['landau_hp', {'P_0': 1.e5,
                                              'T_0': 298.15,
                                              'Tc_0': 955.0,
                                              'S_D': 15.6,
                                              'V_D': 0.0}]]

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

2.1.5.3 The CombinedMineral class

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

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

```
Counter({'O': np.float64(6.0), 'Si': np.float64(2.0), 'Mg': np.float64(1.0), 'Fe
↪': np.float64(1.0)})
```

2.1.5.4 Implemented datasets

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

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

[7]:
```
from burnman import minerals
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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

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

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

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

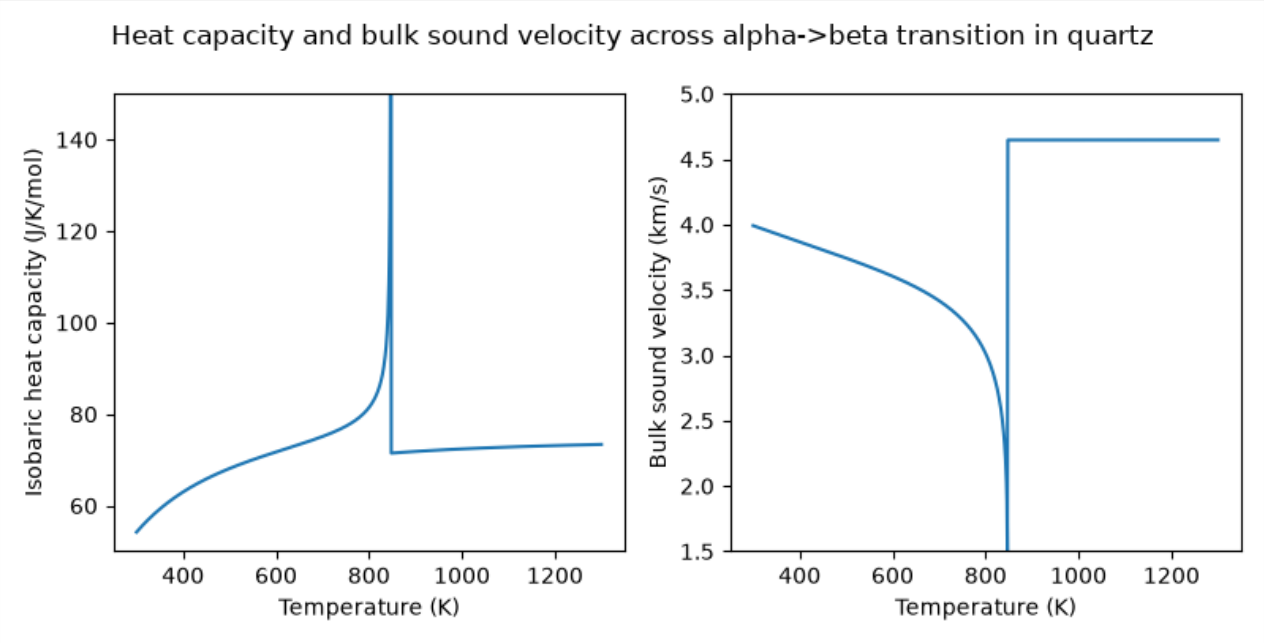
[10]:
```
# Necessary imports for creating the P, T arrays and plotting
import numpy as np
import matplotlib.pyplot as plt

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

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

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

(continues on next page)

(continued from previous page)

```
print(qtz.params)
print(qtz.property_modifiers)

# Return arrays containing the desired properties
isobaric_heat_capacities, bulk_sound_velocities = qtz.evaluate(properties,␣
↪pressures, temperatures)

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

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

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



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

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

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

2.1.6 Solution objects

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

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

2.1.6.1 Model formalisms

The ideal model

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

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

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

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

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

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

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

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

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

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

[13]:
```
CMS_melt = Solution(name = 'CMAS melt (ideal approximation)',
                    solution_model = IdealSolution(endmembers = [[minerals.HGP_
↪2018_ds633.q4L,
                                                                  "[]0[Sinet]"],
                                                                  [minerals.HGP_
↪2018_ds633.wo1L,
```

(continues on next page)

(continued from previous page)

```
                                                                    "[Ca][Sichain]
↪"],
                                                                    [minerals.HGP_
↪2018_ds633.fo2L,
                                                                    "[Mg]4[Sitet]
↪"]]
                                                                ),
                    molar_fractions = [0.1, 0.4, 0.5])
```

In the endmember formulae above, the first site has different species (absent, Ca, Mg) *and* a different multiplicity (0, 1, 4) for the three endmembers. On the second site, the multiplicity is constant with composition, but Si is treated as three different species (network, chain and isolated tetrahedra).

The (a)symmetric model

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

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

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

and

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

where

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

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

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

g2 = Solution(name='asymmetric garnet (ThermoCalc ds6.2)',
              solution_model=AsymmetricRegularSolution(endmembers=[[minerals.HP_
↪2011_ds62.py(), '[Mg]3[Al]2Si3O12'],
                                                                   [minerals.HP_
↪2011_ds62.alm(), '[Fe]3[Al]2Si3O12'],
                                                                   [minerals.HP_
↪2011_ds62.gr(), '[Ca]3[Al]2Si3O12'],
                                                                   [minerals.HP_
↪2011_ds62.andr(), '[Ca]3[Fe]2Si3O12']],
                                                        alphas = [1.0, 1.0, 2.7,␣
↪2.7],
```

(continues on next page)

(continued from previous page)

```
                                                    energy_interaction = [[2.
↪5e3, 31.e3, 53.2e3],
                                                                          [5.
↪e3, 37.24e3],
                                                                          [2.
↪e3]]))
```

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

The subregular model

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

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

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

[15]:
```
from burnman.classes.solutionmodel import SubregularSolution

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

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

energy_interaction = add(mult(Wh_1bar, 3.), mult(Wv, -3.e5))
volume_interaction = mult(Wv, 3.)
entropy_interaction = mult(Ws, 3.)

g3 = Solution(name='CFMnMAS garnet (Ganguly et al., 1996)',
              solution_model=SubregularSolution(
              endmembers=[[minerals.HP_2011_ds62.py(), '[Mg]3[Al]2Si3O12'],
```

(continues on next page)

(continued from previous page)

```
                        [minerals.HP_2011_ds62.alm(), '[Fe]3[Al]2Si3O12'],
                        [minerals.HP_2011_ds62.gr(), '[Ca]3[Al]2Si3O12'],
                        [minerals.HP_2011_ds62.spss(), '[Mn]3[Al]2Si3O12']],
              energy_interaction=energy_interaction,
              volume_interaction=volume_interaction,
              entropy_interaction=entropy_interaction,
              energy_ternary_terms = [[0, 1, 2, 0.e3]],
              volume_ternary_terms = [[0, 1, 2, 0.e-6]],
              entropy_ternary_terms = [[0, 1, 2, 0.]]))
```

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

2.1.6.2 Implemented datasets

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

[16]:
```
bdg = minerals.SLB_2011.mg_fe_bridgmanite()
bdg.endmembers
```

[16]:
```
[[<burnman.minerals.SLB_2011.mg_perovskite at 0x74653aa64830>, '[Mg][Si]O3'],
 [<burnman.minerals.SLB_2011.fe_perovskite at 0x74653aa64590>, '[Fe][Si]O3'],
 [<burnman.minerals.SLB_2011.al_perovskite at 0x74653aa64980>, '[Al][Al]O3']]
```

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

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

[17]:
```
bdg.set_composition([0.8, 0.1, 0.1])
bdg.set_state(30.e9, 2000.)

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

```
Bridgmanite composition: {'Mg': np.float64(0.8), 'Si': np.float64(0.9), 'O': np.
↪float64(3.0), 'Fe': np.float64(0.1), 'Al': np.float64(0.2)}
Bridgmanite volume: 2.315e-05 m
Bridgmanite endmember partial Gibbs energies:
```

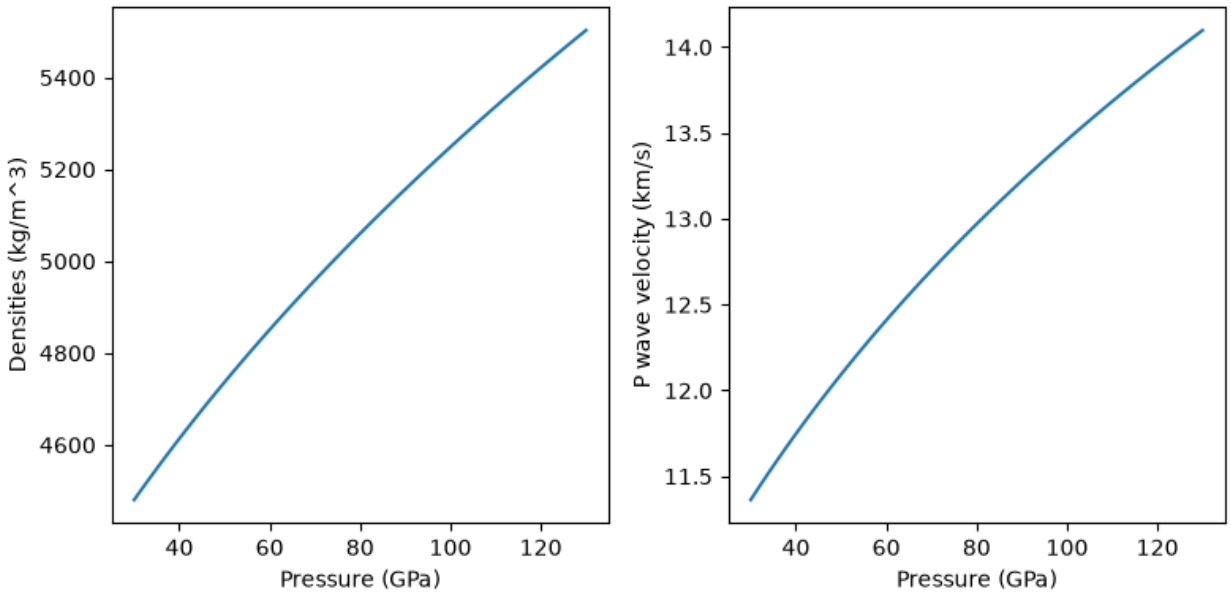
(continues on next page)

(continued from previous page)

```
   -9.690e+02 kJ/mol
   -6.868e+02 kJ/mol
   -1.143e+03 kJ/mol
```

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

[18]:
```
pressures = np.linspace(30.e9, 130.e9, 101)
temperatures = 2000.*np.ones_like(pressures)
densities, p_wave_velocities = bdg.evaluate(['rho', 'v_p'], pressures,␣
↪temperatures)

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

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

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



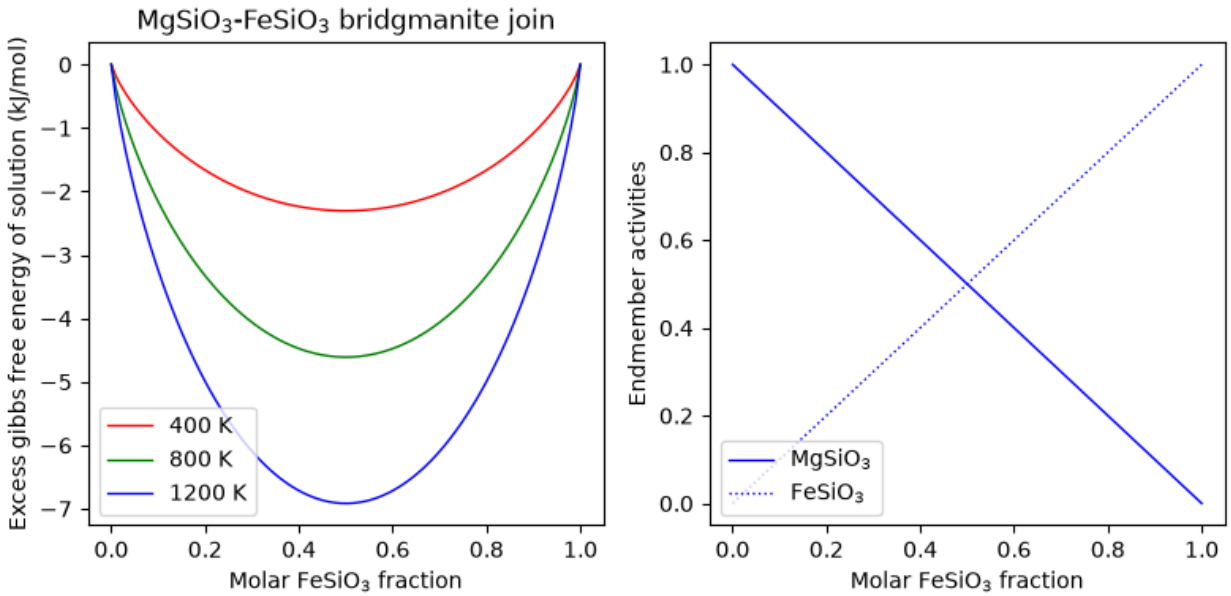
The “set_composition” method must be used every time the Solution composition is to be changed:

[19]:
```
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

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

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

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

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

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

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


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

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

fig.set_layout_engine('tight')
plt.show()
```



2.1.7 Composite objects

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

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

[20]:
```
from burnman import Composite

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

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

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

(continues on next page)

(continued from previous page)

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

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

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

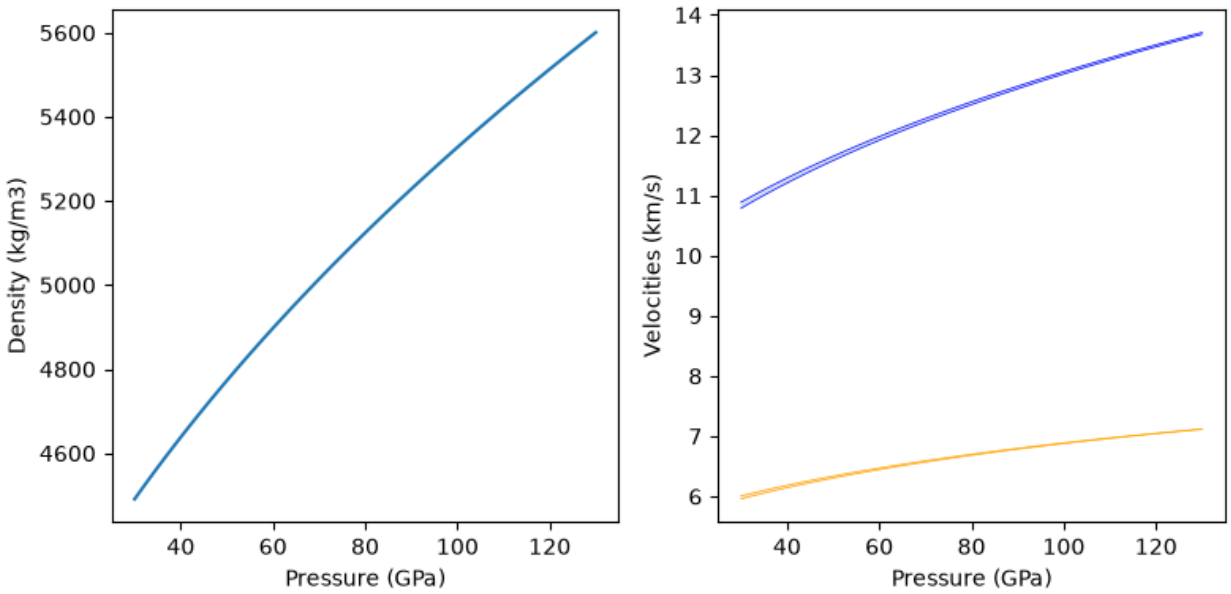
In this code block, we also demonstrate the ability for BurnMan to switch between different schemes for averaging seismic properties. Available schemes are "Voigt", "Reuss", "VoigtReussHill" (the standard scheme, which averages the Voigt and Reuss averages), "HashinShtrikmanLower" and "HashinShtrikmanUpper".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

[21]:
```
pressures = np.linspace(30.e9, 130.e9, 101)
temperatures = np.ones(101)*2000.

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

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

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


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

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

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

fig.set_layout_engine('tight')
```

(continues on next page)

(continued from previous page)

```
plt.show()
```



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

The BurnMan Tutorial

2.2 Part 2: The Composition Class

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

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

2.2.1 Introduction

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

Demonstrates

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

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

2.2.2 The Composition class

It is quite common in petrology to want to perform simple manipulations on chemical compositions. These manipulations might include:

- converting between molar and weight percent of oxides or elements
- changing from one compositional basis to another (e.g. 'FeO' and 'Fe2O3' to 'Fe' and 'O')

- adding new chemical components to an existing composition in specific proportions with existing components.

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

[1]:
```
from burnman import Composition

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

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

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

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

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

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

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

However, this composition is not the composition we wish to make in the lab. We need to make the following changes:

- CaO and Na_2O should be added as $CaCO_3$ and Na_2CO_3.
- FeO should be added as Fe_2O_3

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

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

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

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

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

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

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

The BurnMan Tutorial

2.3 Part 3: Layers and Planets

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

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

2.3.1 Introduction

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

Demonstrates

1. burnman.Layer
2. burnman.Planet

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

2.3.2 Building a planet

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

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

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

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

2.3.3 The Layer class

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

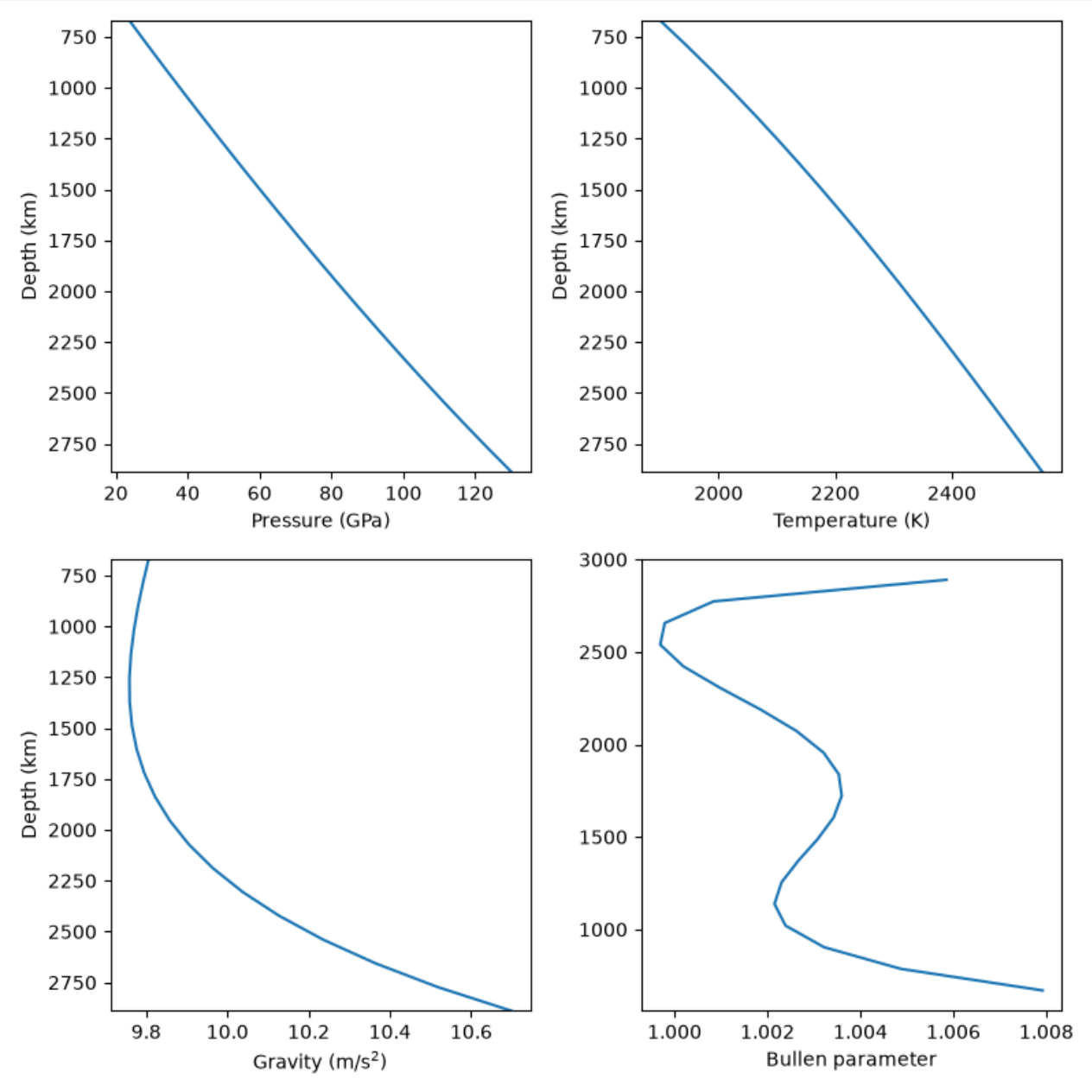
The following code block creates a lower_mantle layer of mg_bridgmanite and periclase in 80:20 molar proportions, and assigns an adiabatic profile with a temperature of 1900 K at the top of the layer. The pressure is set to be self-consistent; i.e. the gravity and pressure are adjusted to ensure that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

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


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

lower_mantle = Layer(name='Lower Mantle', radii=6371.e3-depths)
lower_mantle.set_material(rock)
```

(continues on next page)

(continued from previous page)

```
lower_mantle.set_temperature_mode(temperature_mode='adiabatic',
                                  temperature_top=1900.)
lower_mantle.set_pressure_mode(pressure_mode='self-consistent',
                               pressure_top=23.8e9,
                               gravity_bottom=10.7)

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

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

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

fig.set_layout_engine('tight')
```



2.3.4 The Planet class

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

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

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

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


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

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

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

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

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

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

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

inner_core.set_material(inner_core_material)

inner_core.set_temperature_mode('adiabatic')
```

Now, we create an adiabatic outer core.

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

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

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

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

outer_core.set_material(outer_core_material)

outer_core.set_temperature_mode('adiabatic')
```

Now, we assume that there is a single mantle layer that is convecting. We import a PerpleX input table that contains the material properties of pyrolite for this layer. There are two reasons why we use a PerpleX input table, rather than a native BurnMan composite:

- to allow the planet to have a thermodynamically equilibrated mantle (i.e., one where all of the minerals are stable at the given pressure and temperature). While BurnMan can be used to equilibrate known assemblages using the `burnman.equilibrate()` function, Python is insufficiently fast to calculate equilibrium assemblages on-the-fly.
- to illustrate the coupling with PerpleX.

We apply a perturbed adiabatic temperature profile.

[5]:
```
from burnman import BoundaryLayerPerturbation

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

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

# Import a low resolution PerpleX data table.
```

(continues on next page)

(continued from previous page)

```
fname = '../tutorial/data/pyrolite_perplex_table_lo_res.dat'
pyrolite = PerplexMaterial(fname, name='pyrolite')
convecting_mantle.set_material(pyrolite)

# Here we add a thermal boundary layer perturbation, assuming that the
# lower mantle has a Rayleigh number of 1.e7, and that the basal thermal
# boundary layer has a temperature jump of 840 K and the top
# boundary layer has a temperature jump of 60 K.
tbl_perturbation = BoundaryLayerPerturbation(radius_bottom=cmb_radius,
                                             radius_top=lab_radius,
                                             rayleigh_number=1.e7,
                                             temperature_change=900.,
                                             boundary_layer_ratio=60./900.)

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

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

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

And the lithosphere has a user-defined conductive gradient.

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

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

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

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

(continues on next page)

(continued from previous page)

```
                            np.linspace(moho_temperature,
                                        surface_temperature, 11))
```

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

[8]:
```
planet_zog = Planet('Planet Zog',
                    [inner_core, outer_core,
                     convecting_mantle, lithospheric_mantle,
                     crust], verbose=False)
planet_zog.make()
```

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

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

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

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

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

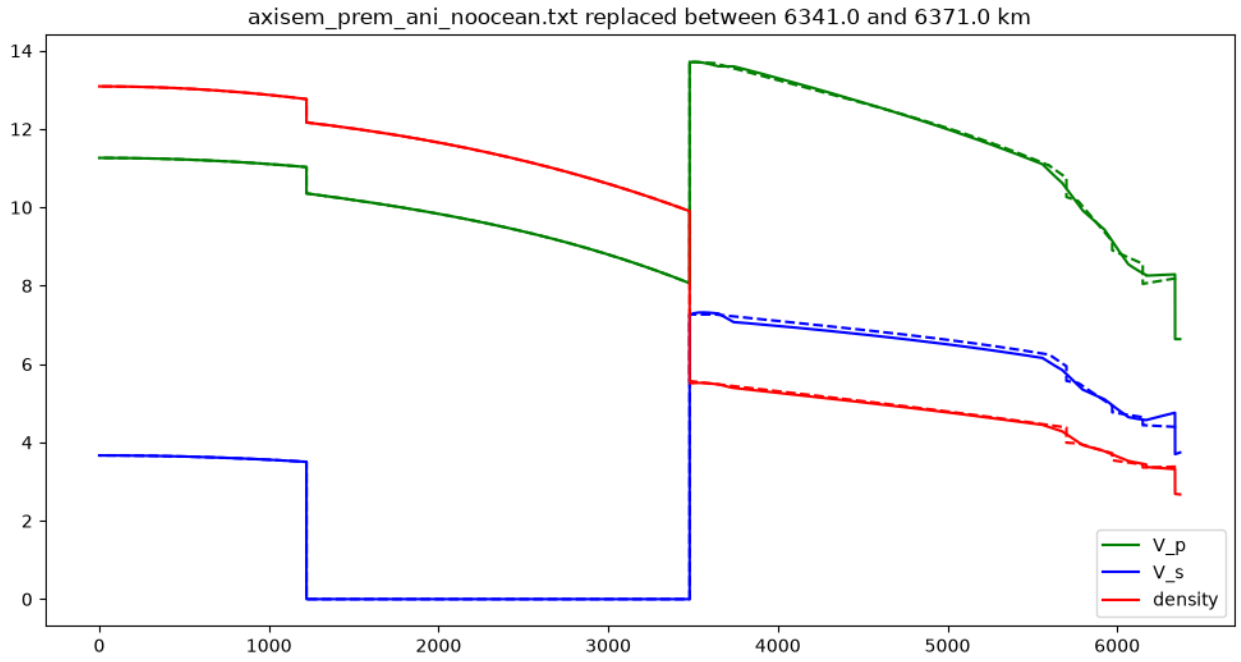
BurnMan contains some utility functions that output data in a format readable by seismic codes such as Axisem and Mineos.

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

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

# Now we delete the newly-created files. If you want them, comment out these␣
↪lines.
import os
```

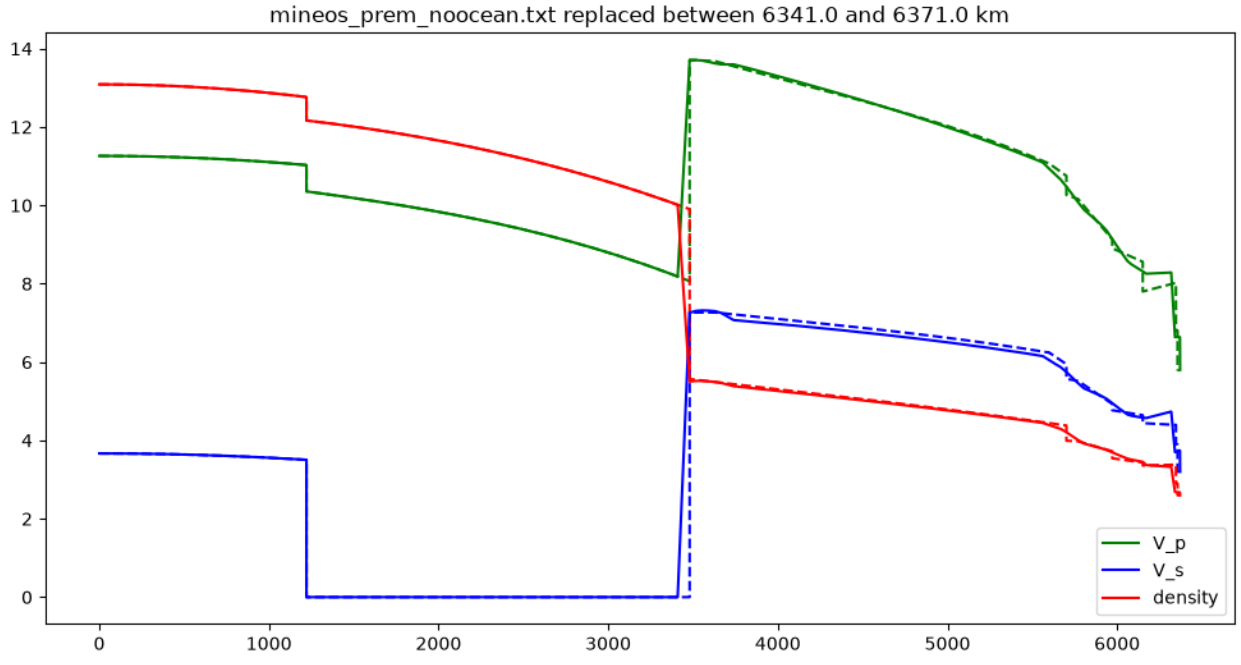
(continues on next page)

(continued from previous page)

```
os.remove('axisem_zog_silicates.txt')
os.remove('mineos_zog_silicates.txt')
```




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




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

Let's compare the properties of this planet to PREM

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

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

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

Finally, plot the 1D structure of the planet

[12]:
```
fig = plt.figure(figsize=(8, 5))
ax = [fig.add_subplot(2, 2, i) for i in range(1, 5)]


bounds = np.array([[layer.radii[0]/1.e3, layer.radii[-1]/1.e3]
                   for layer in planet_zog.layers])
maxy = [15, 400, 12, 7000]
for bound in bounds:
    for i in range(4):
        ax[i].fill_betweenx([0., maxy[i]],
                            [bound[0], bound[0]],
                            [bound[1], bound[1]], alpha=0.2)

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

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

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

# Make a subplot showing the calculated temperature profile
```

(continues on next page)

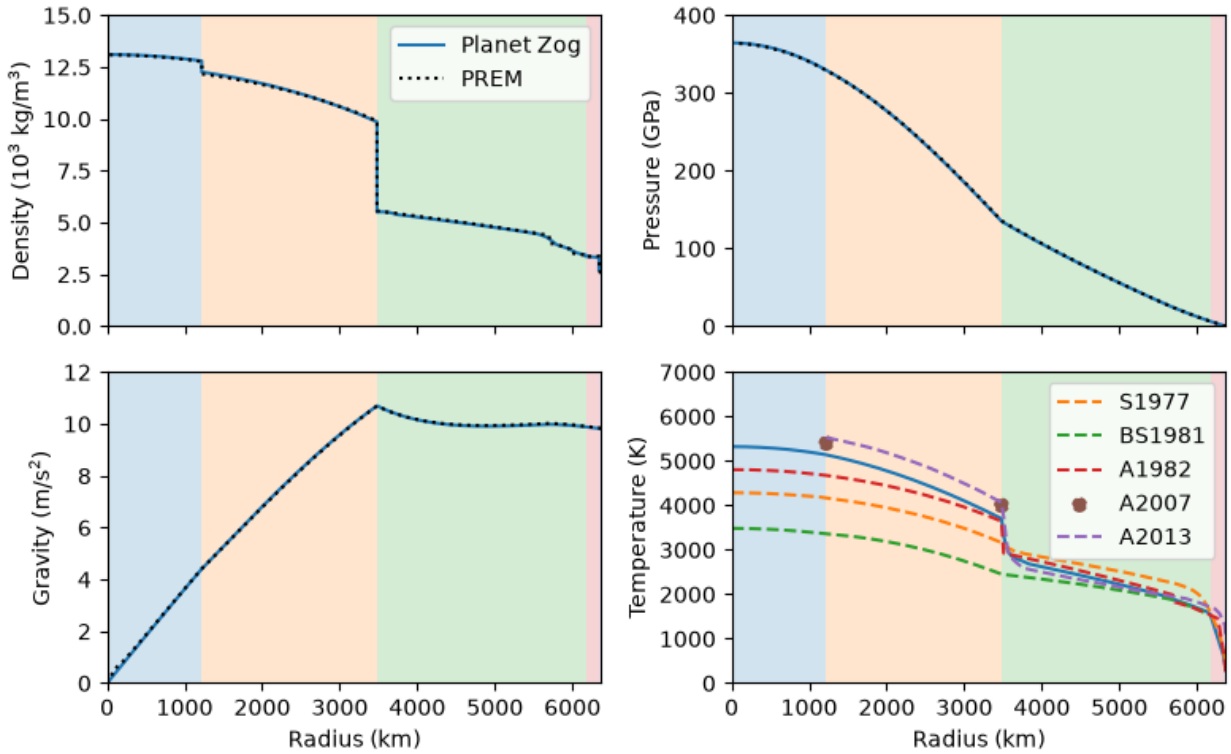
(continued from previous page)

```
ax[3].plot(planet_zog.radii / 1.e3, planet_zog.temperature)
ax[3].set_ylabel('Temperature (K)')
ax[3].set_xlabel('Radius (km)')
ax[3].set_ylim(0.,)

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

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

ax[3].plot(planet_zog.radii / 1.e3,
burnman.geotherm.StaceyContinental().temperatures(planet_zog.depth),
linestyle='--', label=short_labels[0])
mask = planet_zog.depth > 269999.
ax[3].plot(planet_zog.radii[mask] / 1.e3,
           burnman.geotherm.BrownShankland().temperatures(planet_zog.
↪depth[mask]),
           linestyle='--', label=short_labels[1])

ax[3].plot(planet_zog.radii / 1.e3,
           burnman.geotherm.Anderson().temperatures(planet_zog.depth),
           linestyle='--', label=short_labels[2])

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

# Anzellini et al. (2013) geotherm only extends to the ICB
mask = planet_zog.depth < 5200.e3
ax[3].plot(planet_zog.radii[mask] / 1.e3,
           burnman.geotherm.Anzellini2013(prem).temperatures(planet_zog.
↪depth[mask]),
           linestyle='--', label=short_labels[4])

ax[3].legend()
```

(continues on next page)

(continued from previous page)

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

fig.set_layout_engine('tight')
plt.show()
```



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

The BurnMan Tutorial

2.4 Part 4: Fitting

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

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

2.4.1 Introduction

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

Demonstrates

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

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

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

2.4.2 Fitting parameters for an equation of state to experimental data

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

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

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

(continues on next page)

(continued from previous page)

```
                [47.49, 41.280, 0.007]])

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

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

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

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

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

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

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

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

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

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

```
Optimized equation of state for stishovite:
V_0: (1.4005 +/- 0.0001) x 1e-05
```

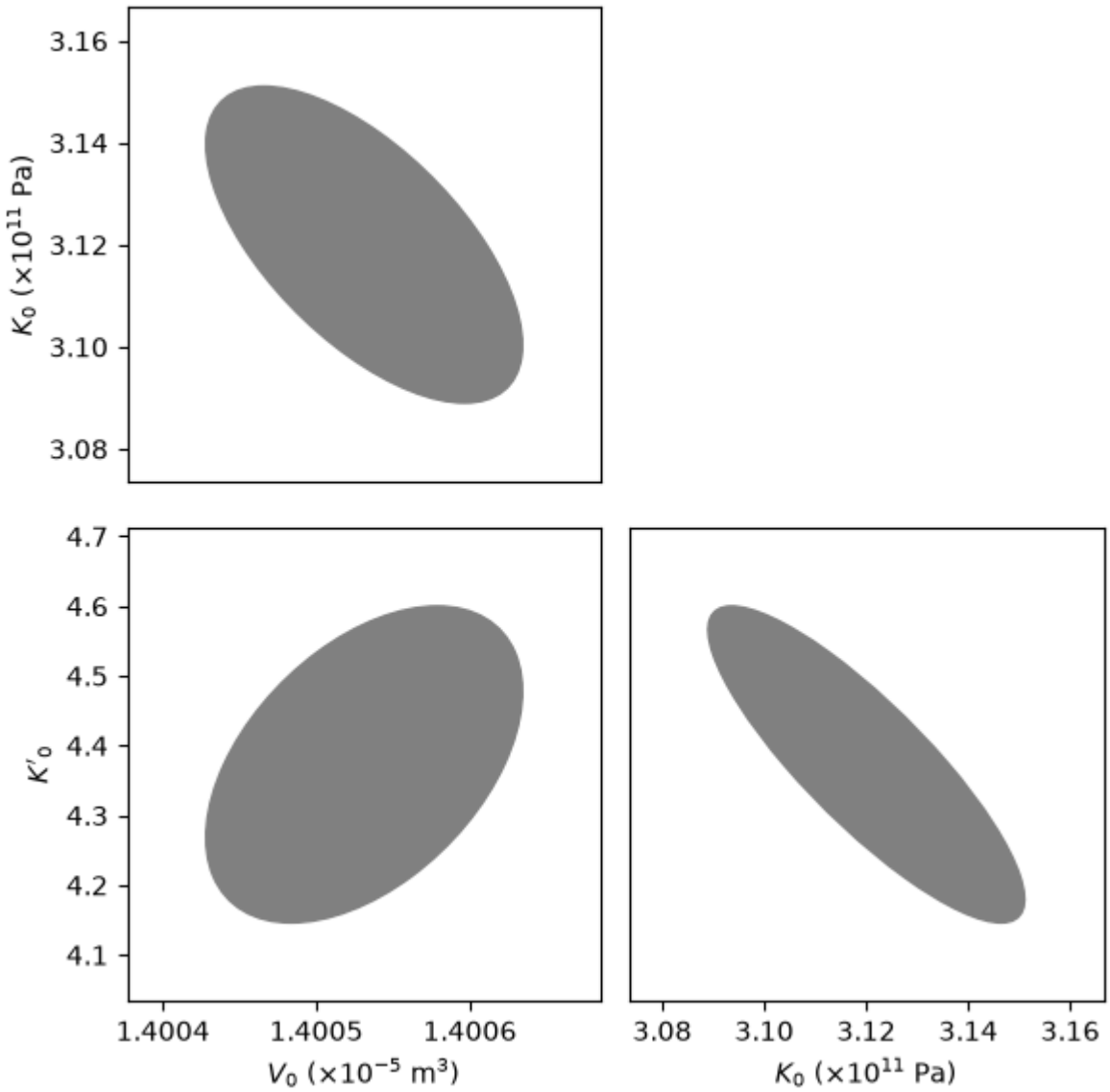
(continues on next page)

(continued from previous page)

```
K_0: (3.12 +/- 0.03) x 1e+11
Kprime_0: (4.4 +/- 0.2) x 1e+00
```

The `fitted_eos` object contains a lot of useful information about the fit. In the next code block, we fit the corner plot of the covariances, showing the tradeoffs in different parameters.

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

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



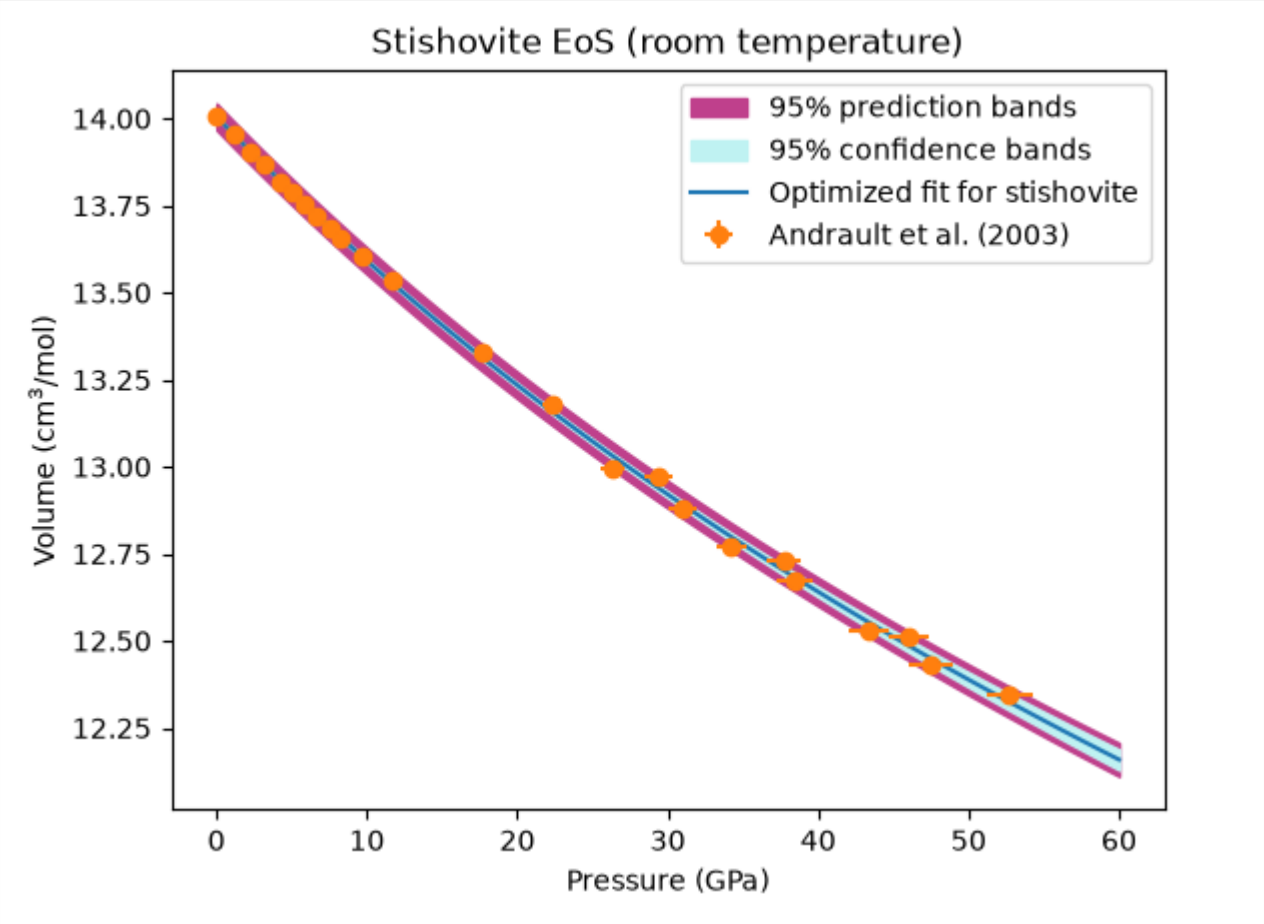
We now plot our optimized equation of state against the original data. BurnMan also includes a useful function `burnman.optimize.nonlinear_fitting.confidence_prediction_bands` that can be used to calculate the nth percentile confidence and prediction bounds on a function given a model using the delta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

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

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

(continues on next page)

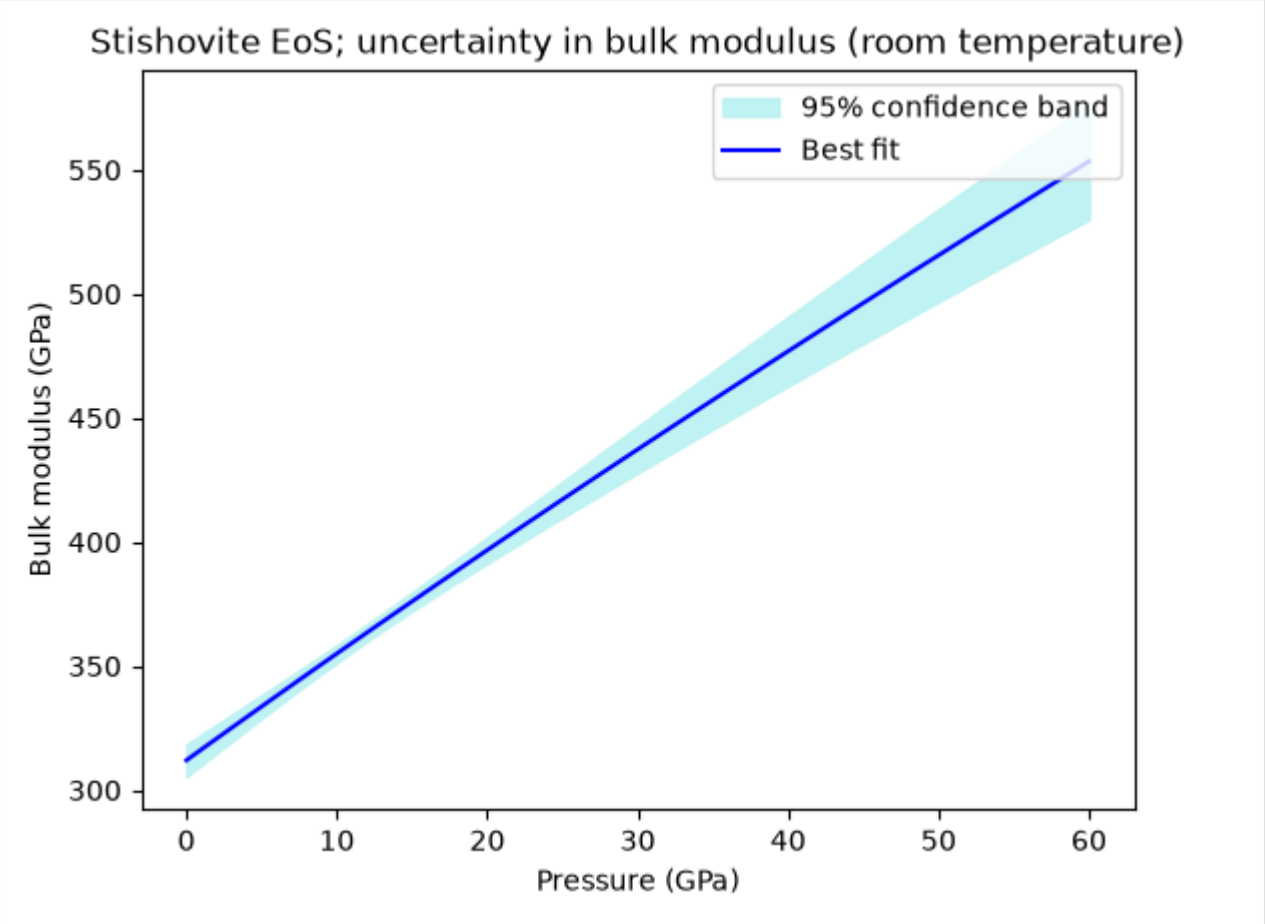
(continued from previous page)

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

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

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

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



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

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

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

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



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

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

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

[8]:
```
from burnman import minerals

gt = minerals.JH_2015.garnet()

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

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

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

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

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

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

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

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

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

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

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

# Print the optimized parameters and principal uncertainties
print('Molar fractions:')
for i in range(len(popt)):
    print(f'{gt.endmember_names[i]}: '
```

(continues on next page)

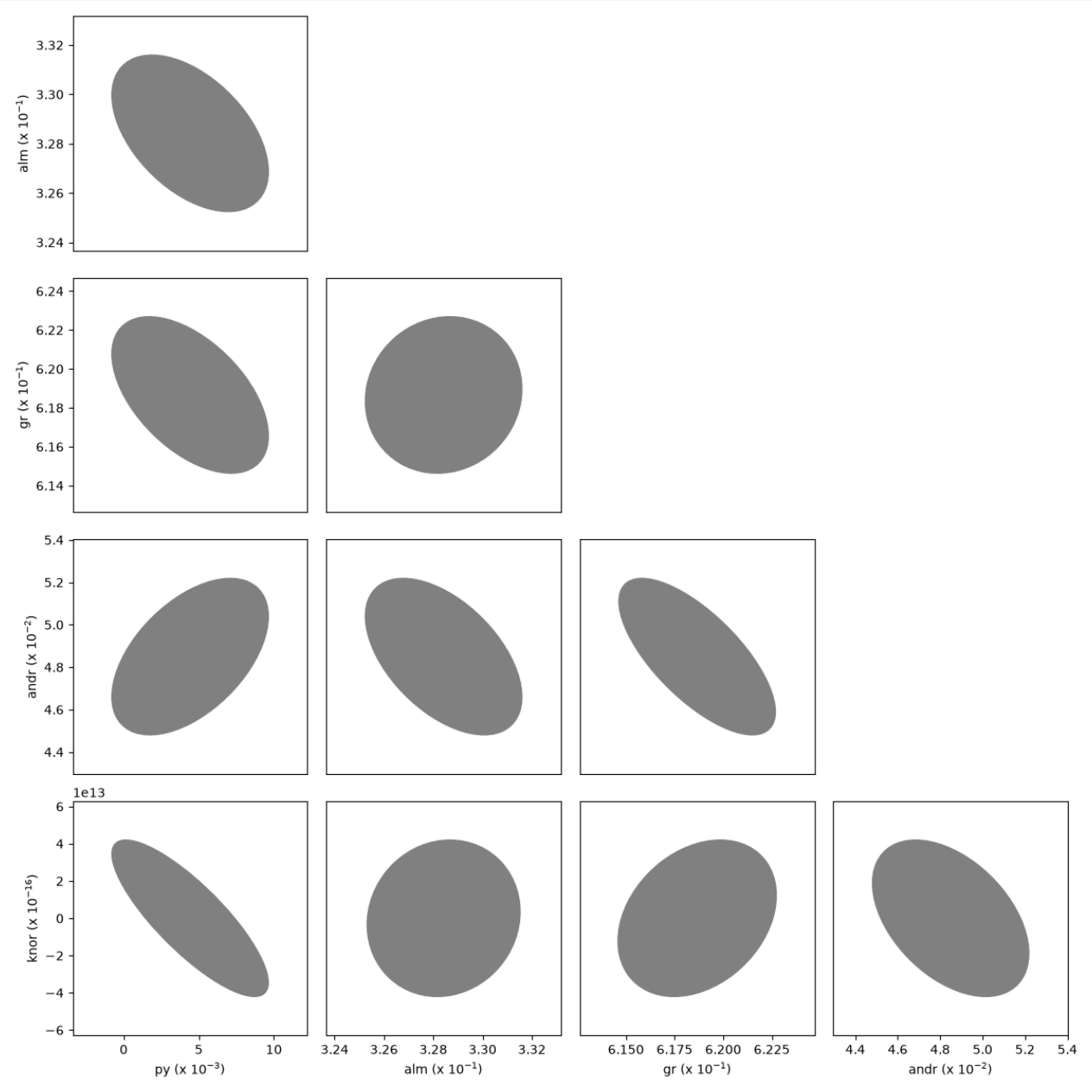
(continued from previous page)

```
            f'{gt.molar_fractions[i]:.3f} +/- '
            f'{np.sqrt(pcov[i][i]):.3f}')

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

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

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

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



2.4.4 Fitting phase proportions to a bulk composition

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

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

First, some tedious data preparation...

[14]:
```
import itertools

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

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

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

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

    residuals.append(res)

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

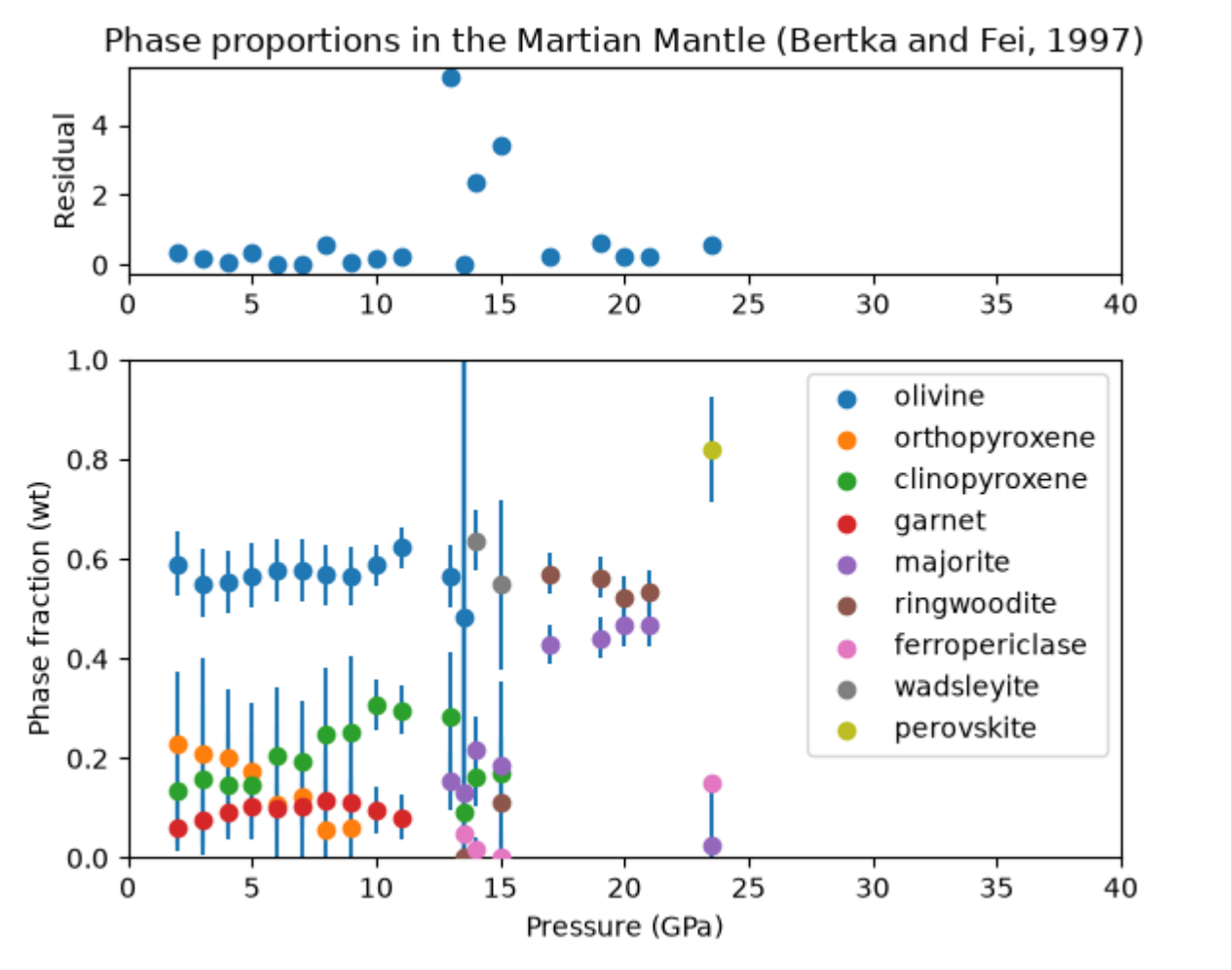
Finally, we plot the data.

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

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

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

(continues on next page)

(continued from previous page)

```
ax[1].legend()
fig.set_layout_engine('tight')
plt.show()
```



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

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

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

The BurnMan Tutorial

2.5 Part 5: Equilibrium problems

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

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

2.5.1 Introduction

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

Demonstrates

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

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

2.5.2 Phase equilibria

2.5.2.1 What BurnMan does and doesn't do

Members of the BurnMan Team are often asked whether BurnMan does Gibbs energy minimization. The short answer to that is no, for three reasons:

1) Python is ill-suited to such computationally intensive problems.

2) There are many pieces of software already in the community that do Gibbs energy minimization, including but not limited to: PerpleX, HeFESTo, Theriak Domino, MELTS, ENKI, FactSAGE (proprietary), and MMA-EoS.

3) Gibbs minimization is a hard problem. The brute-force pseudocompound/simplex technique employed by Perple_X is the only globally robust method, but clever techniques have to be used to make the computations tractable, and the solution found is generally only a (very close) approximation to the true minimum assemblage. More refined Newton / higher order schemes (e.g. HeFESTo, MELTS, ENKI) provide an exact solution, but can get stuck in local minima or even fail to find a solution.

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

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

called `burnman.PerplexMaterial` that is specifically designed to read in and interrogate P-T data from PerpleX.

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

2.5.3 Equilibrating at fixed bulk composition

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

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

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

# The total entropy of the assemblage is the molar entropy
# multiplied by the number of moles in the assemblage. This can be
# accessed either by sol.assemblage.entropy, or by the shorter alias:
entropy = sol.assemblage.S
```

```
It is True that equilibrate was successful
Composite: NCFMAS ol-opx-gt assemblage
  P, T: 1e+10 Pa, 1500 K
Phase and endmember fractions:
  olivine: 0.4971
    Forsterite: 0.9339
    Fayalite: 0.0661
  orthopyroxene: 0.2925
    Enstatite: 0.8640
    Ferrosilite: 0.0687
    Mg_Tschermaks: 0.0005
    Ortho_Diopside: 0.0668
  garnet: 0.2104
    Pyrope: 0.4458
    Almandine: 0.1239
    Grossular: 0.2607
    Mg_Majorite: 0.1258
    Jd_Majorite: 0.0437
```

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

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

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

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

[3]:
```
data = np.array([[sol.assemblage.pressure,
                  sol.assemblage.temperature,
                  sol.assemblage.p_wave_velocity,
                  sol.assemblage.shear_wave_velocity,
                  sol.assemblage.molar_fractions[0],
                  sol.assemblage.molar_fractions[1],
                  sol.assemblage.molar_fractions[2]]
```

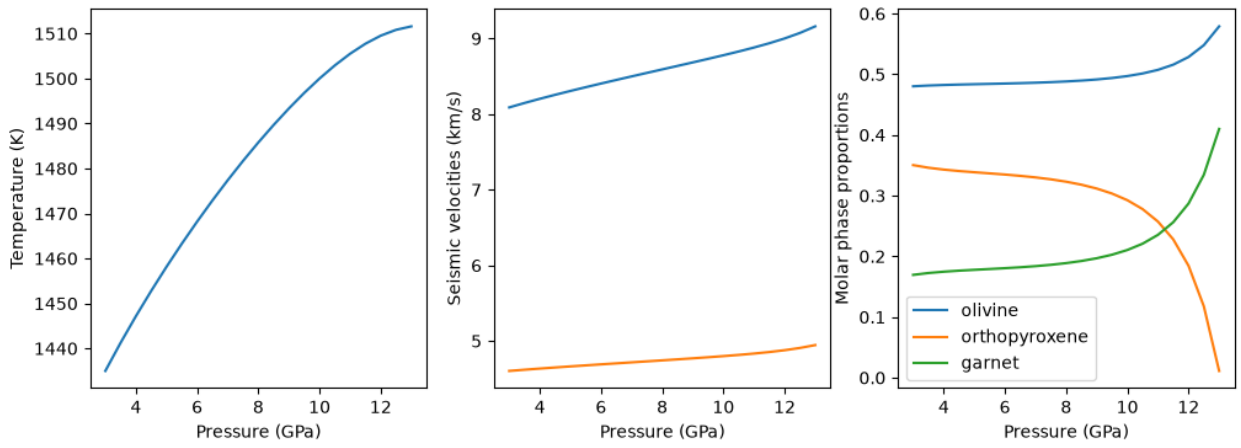
(continues on next page)

(continued from previous page)

```
                    for sol in sols if sol.success])
```

The next code block plots these properties.

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

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

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

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

plt.show()
```



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

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

print(f'Orthopyroxene is exhausted from the assemblage at {sol.assemblage.
```

(continues on next page)

(continued from previous page)

```
↪pressure/1.e9:.2f} GPa, {sol.assemblage.temperature:.2f} K.')
```

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

2.5.4 Equilibrating while allowing bulk composition to vary

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

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

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

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

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

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

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

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

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

[8]:
```
output = []
for (m1, m2, x_fe_m1) in [[ol, wad, np.linspace(x_fe_mbr[0], 0.001, 20)],
                          [ol, rw, np.linspace(x_fe_mbr[0], 0.999, 20)],
                          [wad, rw, np.linspace(x_fe_mbr[1], 0.001, 20)]]:

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

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

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

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

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

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

output = np.array(output)
```

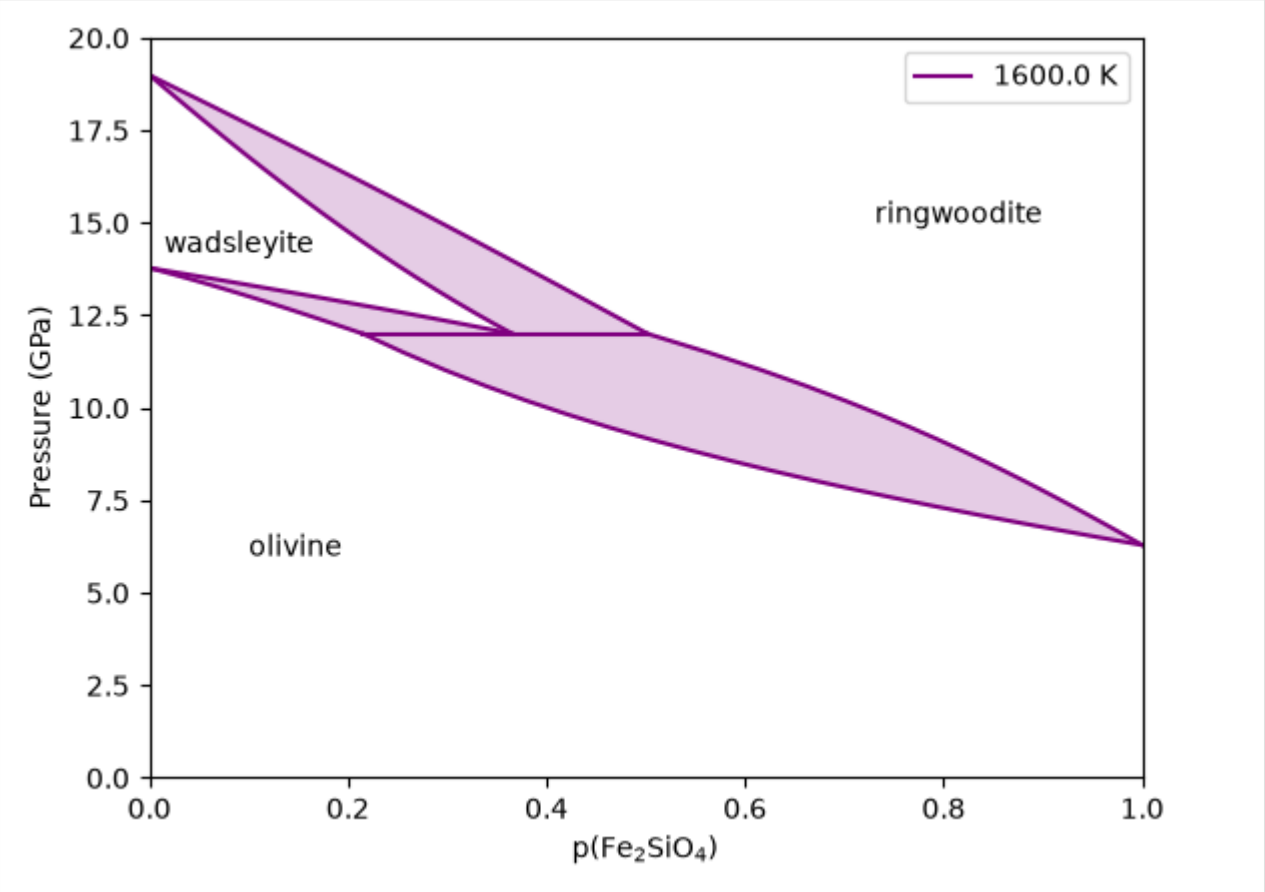
Finally, we do some plotting

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

color='purple'
# Plot the line connecting the three phases
```

(continues on next page)

(continued from previous page)

```
ax[0].plot([x_fe_mbr[0], x_fe_mbr[2]],
           [P_univariant/1.e9, P_univariant/1.e9], color=color)

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

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

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

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



And we're done!

2.6 CIDER Tutorial 2014

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

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

2.6.1 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.

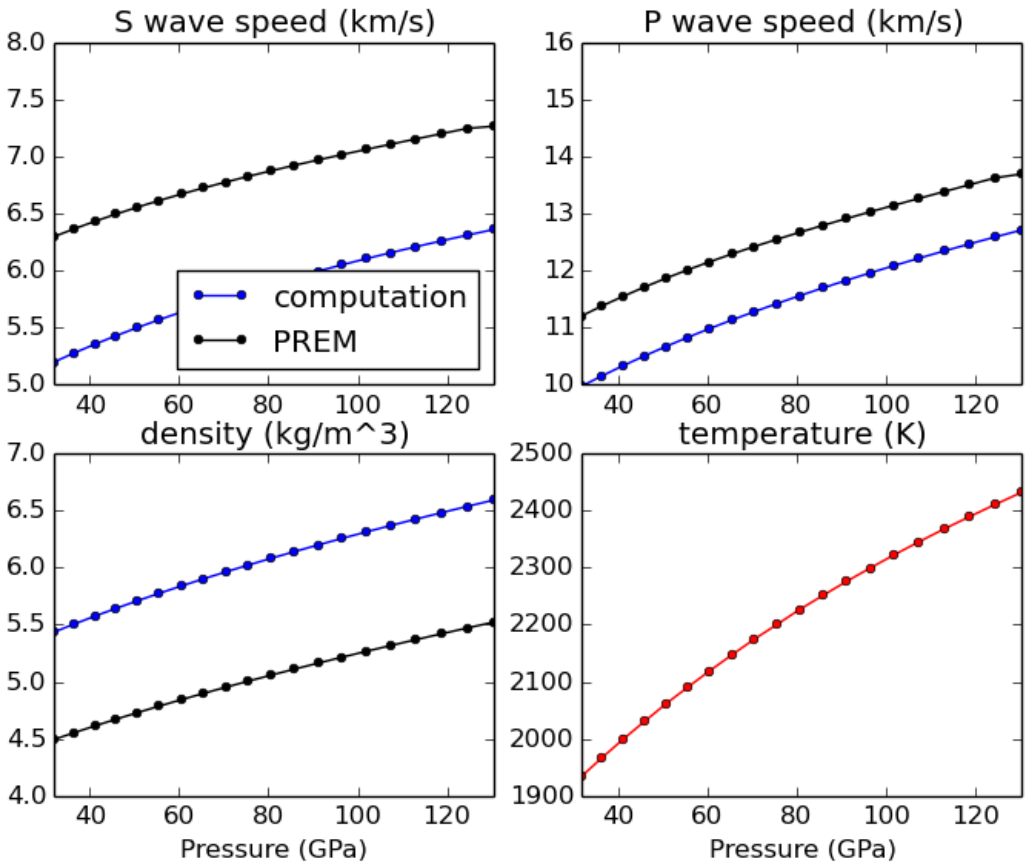
3) Plot those elastic properties, and compare them to a seismic model, in this case PREM

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…

2.6.2 CIDER 2014 BurnMan Tutorial — step 2

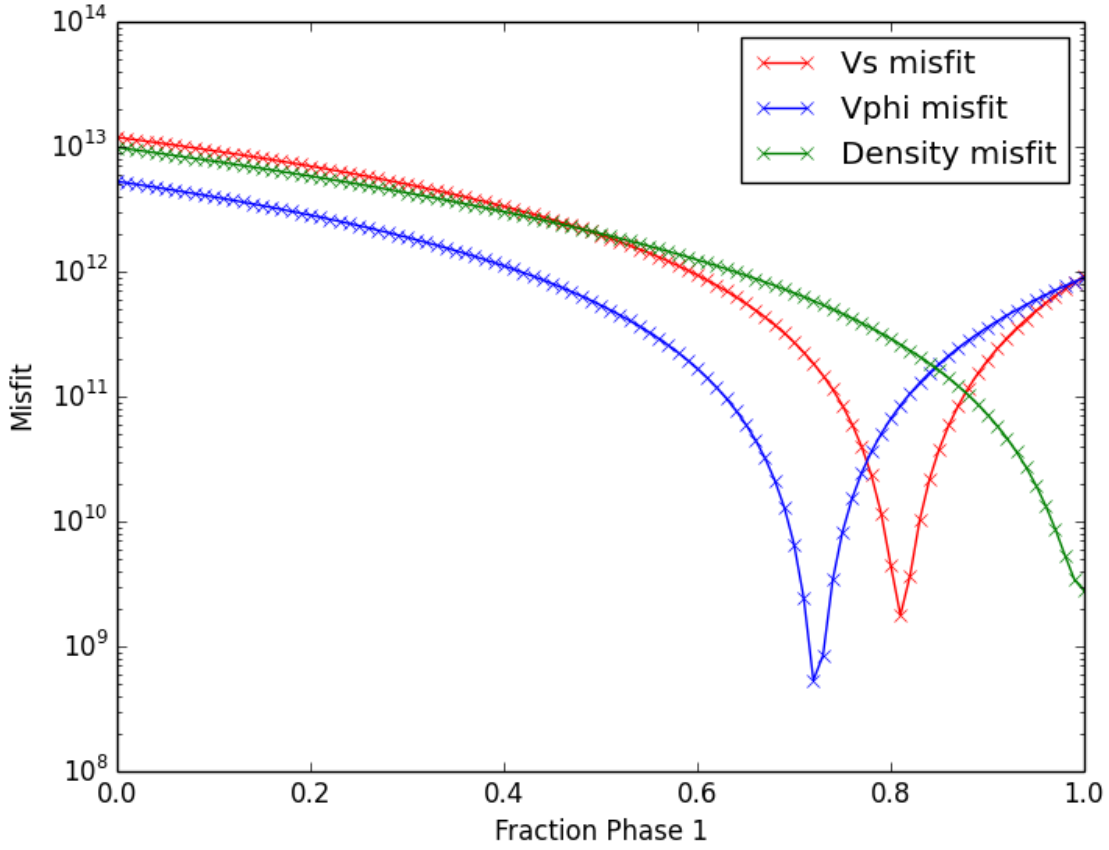
In this second part of the tutorial we try to get a closer fit to our 1D seismic reference model. In the simple Mg, Si, and O model that we used in step 1 there was one free parameter, namely phase_1_fraction, which goes between zero and one.

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:



2.6.3 CIDER 2014 BurnMan Tutorial — step 3

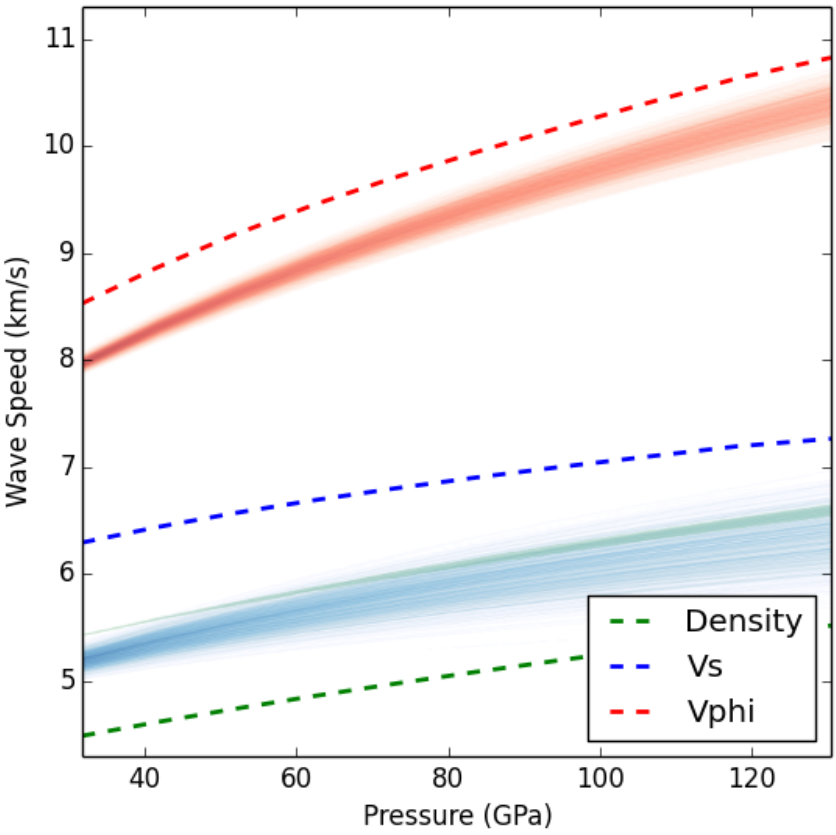
In the previous two steps of the tutorial we tried to find a very simple mineralogical model that best fit the 1D seismic model PREM. But we know that there is consideral uncertainty in many of the mineral physical parameters that control how the elastic properties of minerals change with pressure and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


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

CHAPTER

THREE

MATERIALS

A key feature of BurnMan is its modular treatment of materials. Materials are represented by classes that define their thermodynamic and seismic properties.

3.1 Material Base Class

3.1.1 Overview

The *Material* class serves as the **base class** for all materials in BurnMan, including *Mineral Classes* (*Mineral*), *Solutions* (*Solution*), and *Composite Class* (*Composite*). It defines the common interface through which all thermodynamic and elastic properties are computed, as well as the workflow for setting state variables and selecting thermodynamic models.

Subclasses implement the specific physical behaviour (equations of state, solution models, mechanical mixing laws), while *Material* ensures that all materials (simple or composite) expose a consistent and unified interface.

Regardless of type, materials support the same set of state variables and property queries.

Typical usage follows this pattern:

1. **Instantiate** a material object.
2. **Set the state** using one or more of:
    - *Material.set_state()* — set pressure and temperature,
    - *Material.number_of_moles* — specify the material amount.
3. **Query thermodynamic properties**, such as density, heat capacities, expansivity, elastic parameters, or seismic velocities.

Thermodynamic properties are computed on demand, with caching to optimise performance. Changing any state variable automatically resets cached values to ensure consistency.

3.1.2 Property Evaluation

A large number of physical properties can be evaluated. Examples include:

- Pressure (`pressure`)
- Temperature (`temperature`)

- Number of moles (`number_of_moles`)
- Mass (`mass`)
- Gibbs energy (`gibbs`)
- Helmholtz energy (`helmholtz`)
- Enthalpy (`enthalpy`)
- Internal energy (`internal_energy`)
- Entropy (`entropy`)
- Density (`density`)
- Volume (`volume`) and molar volume (`molar_volume`)
- Heat capacities (`molar_heat_capacity_p` and `molar_heat_capacity_v`)
- Thermal expansivity (`thermal_expansivity`)
- Gruneisen parameter (`gruneisen_parameter`)
- Isothermal bulk modulus (Reuss) (`isothermal_bulk_modulus_reuss`)
- Isentropic bulk modulus (Reuss) (`isentropic_bulk_modulus_reuss`)
- Corresponding compressibilities (`isothermal_compressibility_reuss`, `isentropic_compressibility_reuss`)
- Shear modulus (`shear_modulus`)
- Effective seismic bulk and shear moduli (`effective_isentropic_bulk_modulus`, `effective_shear_modulus`), see *Averaging Schemes*.
- Seismic velocities (`p_wave_velocity`, `shear_wave_velocity`, `bulk_sound_velocity`), see *Averaging Schemes*.

Molar quantities are available via `molar_`-prefixed properties. Aliases exist for common properties (e.g., `C_p` for `heat_capacity_p`).

Properties generally require that the state (pressure and temperature) has been set before computation.

3.1.3 State Management

`Material.set_state`(*pressure*, *temperature*)

Sets the thermodynamic state of the material by updating its **pressure** (Pa) and **temperature** (K).

`Material.set_state_with_volume`(*volume*, *temperature*)

Sets the state of the material using **volume** (m^3) rather than **pressure** (Pa). The method internally performs a pressure inversion: it solves for the pressure that produces the specified volume at the given temperature.

3.1.4 Introspection and Debugging

`Material.`**`to_string`**`()`

Returns a human-readable string describing the material. Subclasses typically override this to include formula, composition, or model information.

`Material.`**`debug_print`**`()`

Prints detailed diagnostic information about the material, including its current state, cached properties, and any internal solver or model details relevant to the subclass. Intended primarily for debugging complex materials such as solutions or composites.

`Material.`**`unroll`**`()`

Expands the material into a flat list of *burnman.Mineral* objects and their molar fractions.

`Material.`**`print_minerals_of_current_state`**`()`

Prints a formatted list of all constituent minerals after applying the current pressure, temperature, and composition. This method invokes the same internal expansion logic as *Material.unroll()* and is useful for inspecting the active mineral assemblage within solutions or composites.

3.1.5 Evaluation Routines

`Material.`**`evaluate`**`(`*property_list*, *pressures*, *temperatures*, *molar_fractions=None*`)`

Computes and returns a thermodynamic property or properties at the requested conditions. The list of properties should be provided as a list of strings in `property_list`. The conditions are specified via arrays which may be of arbitrary shape, with broadcasting applied as needed. Pressure and temperature arrays must have the same shape.

`Material.`**`evaluate_with_volumes`**`(`*property_name*, *volumes*, *temperatures*, *molar_fractions=None*`)`

Evaluates a specified property using externally supplied **volume** values rather than pressures. This method performs a pressure inversion internally to determine the corresponding pressures for the given volumes at the specified temperatures.

3.2 Mineral Classes

The Mineral classes in BurnMan are intended to represent pure phases with fixed chemical compositions and structures. They form the foundation for more complex materials such as solutions and multi-phase composites.

In this section, we describe the base *Mineral Base Class* (*Mineral*) class and its key features, the *Combined Mineral Class* (*CombinedMineral*) subclass, which allows combining multiple minerals into a single effective phase, the various implemented *Equations of State for Minerals*, and *Mineral Property Modifiers* that can be applied to adjust mineral properties.

3.2.1 Mineral Base Class

The *Mineral* class is the **base class** for all mineral phases in BurnMan. It derives from the *Material Base Class* (*Material*), using an equation of state to implement functionality specific to pure mineral phases with fixed chemical compositions and crystal structures.

3.2.1.1 Key Features

- **Equations of State**: Minerals use a variety of implemented equations of state (EOS) to model their thermodynamic behaviour under varying pressure and temperature conditions. The EOS is selected at instantiation and governs property calculations.
- **Property Modifiers**: Minerals can apply property modifiers to adjust baseline properties from the EOS. This allows incorporation of experimental corrections or additional physical effects.
- **Composition**: Each *Mineral* instance has a fixed chemical composition defined at creation. This composition influences density, molar volume, and other derived properties.

3.2.1.2 Mineral Datasets

BurnMan includes a database of predefined mineral phases with commonly used equations of state and compositions. See *Mineral Datasets* for a list of the available mineral models.

3.2.1.3 Instantiation

Mineral objects are instantiated via a constructor that takes a dictionary of parameters as argument. The parameters required depend on the selected equation of state. Instantiation looks like this:

```
from burnman import Mineral

params = {
    'equation_of_state': 'slb3',  # using the Stixrude and Lithgow-Bertelloni EoS
    # EOS-specific parameters...
}

property_modifiers = [
    ['linear', {'delta_E': 1.e3, 'delta_S': 0., 'delta_V': 0.}],
    # Additional modifiers...
]

mineral = Mineral(params, property_modifiers)
```

An important parameter in the *params* dictionary is the *equation_of_state* key, which specifies the EOS to use. This can either be a string referring to a built-in EOS, or a custom EOS object implementing the required interface. Available equations of state, and their string identifiers, are described in *Equations of State for Minerals*.

Property modifiers are optional and can be omitted. See *Mineral Property Modifiers* for details.

3.2.2 Combined Mineral Class

The *CombinedMineral* class is a subclass of *Mineral* that allows combining multiple mineral phases into a single effective phase. This is useful, for example, when there is not enough information to constrain a complete equation of state for a mineral.

The essential idea is to represent the desired mineral as a weighted mechanical mixture of other minerals with well-defined equations of state. In practice, it is usually desirable to combine minerals that are similar

in structure and composition to the target mineral. If the composite minerals are picked carefully enough, it may be that only a linear correction to the Gibbs energy of the combined phase (as a function of pressure and temperature) is sufficient to reproduce experimental data for the target mineral.

The *CombinedMineral* class takes a list of component minerals and their corresponding molar amounts, along with optional energy adjustments, and computes the overall thermodynamic properties self-consistently. The linear adjustments to the Gibbs energy are given in the order: (1) internal energy [J/mol], (2) entropy [J/(mol·K)], and (3) volume [m^3/mol]:

```
from burnman import CombinedMineral

component_minerals = [mineral1, mineral2, mineral3]
molar_amounts = [0.5, 0.3, 0.2]
energy_adjustments = [1000.0, -5.0, 1.e-6]

combined_mineral = CombinedMineral(
    component_minerals,
    molar_amounts,
    energy_adjustments=energy_adjustments
)
```

In practice, it is usually desirable to combine minerals that are similar in structure and composition to the target mineral. If the composite minerals are picked carefully enough, it may be that only a linear correction to the Gibbs energy of the combined phase (as a function of pressure and temperature) is sufficient to reproduce experimental data for the target mineral. More complex modifications to the Gibbs energy can be achieved by applying *Mineral Property Modifiers* to one or more of the component minerals.

3.2.3 Equations of State for Minerals

To calculate the thermodynamic and thermoelastic properties of a material at a given pressure (P) and temperature (T), a formulation is required that describes how the Gibbs energy (or any of the other three thermodynamic potentials) varies with pressure and temperature. These formulations are known as equations of state (EoS). In BurnMan, each mineral is associated with an EoS, and the mineral knows which EoS to use via a key called "equation_of_state" in its parameter dictionary (this dictionary is passed as an argument when creating the mineral). The value of this key can either be the EoS class itself, an instance of the EoS class, or a string that identifies the EoS. The valid string keys are listed below.

Currently the following EoSs are supported in BurnMan:

- *Isothermal Equations of State*
  - *3rd-order Birch-Murnaghan EoS (Birch, 1947) (bm3)*
  - *3rd-order Birch-Murnaghan with 2nd order Shear Modulus expansion (Birch, 1947) (bm3shear2)*
  - *4th-order Birch-Murnaghan EoS (Birch, 1947) (bm4)*
  - *MACAW EoS (Lozano and Aslam, 2022) (macaw)*

  - *Modified Tait EoS (Huang and Chow, 1974) (mt)*
  - *Morse Potential EoS (Morse, 1929) (morse)*
  - *Murnaghan EoS (Murnaghan, 1944) (murnaghan)*
  - *Reciprocal K' EoS (Stacey and Davis, 2004) (rkprime)*
  - *SPOCK EoS (Myhill, 2025) (spock)*
  - *Vinet EoS (Vinet et al., 1987) (vinet)*
- *Thermal Equations of State*
  - *Liquid Metal EoS (Anderson and Ahrens, 1994) (aa)*
  - *Brosh CALPHAD EoS (Brosh, 2007) (brosh_calphad)*
  - *CORK EoS (Holland and Powell, 1991) (cork)*
  - *De Koker and Stixrude Liquid EoS (de Koker and Stixrude, 2013) (dks_l)*
  - *De Koker and Stixrude Solid EoS (de Koker and Stixrude, 2013) (dks_s)*
  - *Holland and Powell 1998 EoS (Holland and Powell, 1998) (hp98)*
  - *Holland and Powell Modified Tait Thermal EoS (Holland and Powell, 2011) (hp_tmt)*
  - *Holland and Powell Modified Tait Thermal Liquid EoS (Holland and Powell, 2011) (hp_tmtL)*
  - *Mie-Gruneisen-Debye 2nd-order EoS (Matas et al., 2007) (mgd2)*
  - *Mie-Gruneisen-Debye 3rd-order EoS (Matas et al., 2007) (mgd3)*
  - *Modular Mie-Gruneisen-Debye EoS (modular_mgd)*
  - *Modular Mie-Gruneisen-Debye EoS with Anharmonicity (modular_mgd_with_anharmonicity)*
  - *Stixrude and Lithgow-Bertelloni 2nd-order EoS (Stixrude and Lithgow-Bertelloni, 2005) (slb2)*
  - *Stixrude and Lithgow-Bertelloni 3rd-order EoS (Stixrude and Lithgow-Bertelloni, 2005) (slb3)*
  - *Stixrude and Lithgow-Bertelloni 3rd-order Conductive EoS (Stixrude and Lithgow-Bertelloni, 2024) (slb3-conductive)*

These EoSes are described in more detail below.

3.2.3.1 Isothermal Equations of State

Isothermal equations of state describe the relationship between pressure and volume at a constant temperature. BurnMan includes several isothermal equations of state.

3rd-order Birch-Murnaghan EoS (Birch, 1947) (bm3)

class burnman.eos.birch_murnaghan.**BM3**

Bases: *BirchMurnaghanBase*

The third order Birch-Murnaghan isothermal equation of state. The negative finite-strain (or compression) is defined as

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

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:

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

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

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

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.

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy	J/mol
`P_0`	Reference pressure	Pa
`V_0`	Reference volume	m^3
`K_0`	Reference isothermal bulk modulus	Pa
`Kprime_0`	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
`G_0`	Reference shear modulus	Pa
`Gprime_0`	Pressure derivative of the shear modulus at the reference state	Dimensionless

3rd-order Birch-Murnaghan with 2nd order Shear Modulus expansion (Birch, 1947) (bm3shear2)

class burnman.eos.birch_murnaghan.**BM3Shear2**

Bases: *BirchMurnaghanBase*

The third order Birch-Murnaghan isothermal equation of state with second order expansion for the shear modulus. Do not use this unless you have a good reason to.

The negative finite-strain (or compression) is defined as

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

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).

For an isotropic material one obtains for the pressure, isothermal bulk modulus, and shear modulus:

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

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

$$G = \left(1+2f\right)^{5/2}\Big[G_0 + \left(3K_0G_0' - 5G_0\right)f\Big].$$

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.

Parameter	Description	Units
F_0	Reference Helmholtz free energy	J/mol
P_0	Reference pressure	Pa
V_0	Reference volume	m^3
K_0	Reference isothermal bulk modulus	Pa
Kprime_0	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
G_0	Reference shear modulus	Pa
Gprime_0	Pressure derivative of the shear modulus at the reference state	Dimensionless

4th-order Birch-Murnaghan EoS (Birch, 1947) (bm4)

class burnman.eos.birch_murnaghan.**BM4**

Bases: *BirchMurnaghanBase*

Base class for the isothermal Birch Murnaghan equation of state. This is fourth order in strain, and has no temperature dependence.

Note that unlike the third order Birch-Murnaghan equation of state, the shear modulus is not defined for this equation of state.

Parameter	Description	Units
F_0	Reference Helmholtz free energy	J/mol
P_0	Reference pressure	Pa
V_0	Reference volume	m^3
K_0	Reference isothermal bulk modulus	Pa
Kprime_0	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
Kprime_prime_0	Second pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless

MACAW EoS (Lozano and Aslam, 2022) (macaw)

class burnman.eos.macaw.**MACAW**

Bases: *IsothermalEquationOfState*

Class for the MACAW equation of state detailed in [LA22].

This equation of state has no temperature dependence.

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of the bulk modulus at reference pressure.	Dimensionless
Kprime_inf	Infinite pressure derivative of the bulk modulus.	Dimensionless

Modified Tait EoS (Huang and Chow, 1974) (mt)

class burnman.eos.modified_tait.**MT**

Bases: *IsothermalEquationOfState*

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].

$$\frac{V_{P,T}}{V_{1bar,298K}} = 1 - a(1 - (1 + bP)^{-c}),$$

$$a = \frac{1 + K_0'}{1 + K_0' + K_0 K_0''},$$

$$b = \frac{K_0'}{K_0} - \frac{K_0''}{1 + K_0'},$$

$$c = \frac{1 + K_0' + K_0 K_0''}{K_0'^2 + K_0' - K_0 K_0''}$$

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus.	Dimensionless
Kdprime_0	Second pressure derivative of bulk modulus.	Pa^{-1}

Morse Potential EoS (Morse, 1929) (morse)

class burnman.eos.morse_potential.**Morse**

Bases: *IsothermalEquationOfState*

Class for the isothermal Morse Potential equation of state detailed in [StaceyBrennanIrvine81]. This equation of state has no temperature dependence.

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus at zero pressure.	Dimensionless

Murnaghan EoS (Murnaghan, 1944) (murnaghan)

class burnman.eos.murnaghan.**Murnaghan**

Bases: *IsothermalEquationOfState*

The isothermal Murnaghan equation of state, as described in [Mur44].

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus.	Dimensionless

Reciprocal K' EoS (Stacey and Davis, 2004) (rkprime)

class burnman.eos.reciprocal_kprime.**RKprime**

Bases: *IsothermalEquationOfState*

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.

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of bulk modulus at reference conditions.	Dimensionless
`Kprime_inf`	Pressure derivative of bulk modulus at infinite pressure.	Dimensionless

SPOCK EoS (Myhill, 2025) (spock)

class burnman.eos.spock.**SPOCK**

Bases: *IsothermalEquationOfState*

Class for the Scaled Power Of Compression K-prime equation of state. This equation is derived from the assumption that

$$K' = b\left(\frac{V}{V_0}\right)^a$$

This equation of state is described in [Myh25a], but was originally derived by [WMTZ15].

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`P_0`	Reference pressure.	Pa
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of bulk modulus at zero pressure.	Dimensionless
`Kdprime_0`	Second pressure derivative of bulk modulus at zero pressure.	Pa^{-1}
`Kprime_inf`	Infinite pressure limit of the pressure derivative of bulk modulus.	Dimensionless

Vinet EoS (Vinet et al., 1987) (vinet)

class burnman.eos.vinet.**Vinet**

Bases: *IsothermalEquationOfState*

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].

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`P_0`	Reference pressure.	Pa
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of bulk modulus.	Dimensionless

3.2.3.2 Thermal Equations of State

Thermal equations of state account for the effects of temperature on thermoelastic properties. BurnMan includes several thermal equations of state.

Liquid Metal EoS (Anderson and Ahrens, 1994) (aa)

class burnman.eos.aa.**AA**

Bases: *EquationOfState*

Class for the liquid metal EOS detailed in [AndersonAhrens94].

This equation of state is complicated because there is not a single set of independent variables.

The equation of state is based on a reference isentrope passing through a defined volume and entropy point. Internal energy (E) at a given volume is calculated along this isentrope using a fourth order BM EoS (V_0, KS, KS', KS'').

The temperature along the isentrope is calculated via integration of the Grueneisen parameter:

$\gamma = \partial(\ln T)/\partial(\ln \rho)|_S$

which gives:

$T_S/T_0 = \exp(\int(\gamma/\rho)d\rho)$

Finally, the internal energy away from the reference isentrope is calculated as a function of temperature, using an expression for the isochoric heat capacity as a function of volume and temperature.

In this BurnMan implementation, the Helmholtz energy is used as the natural potential, with volume and temperature as the natural variables.

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, the thermal pressure expression is inconsistent with the equation of state away from the principal isentrope.

Note: the expression for Λ (Appendix C) does not reproduce Figure 5. We assume that the figure is correct, and that the correct expression has the form: $F(-325.23 + 302.07(\rho/\rho_0) + 30.45(\rho/\rho_0)^{0.4})$.

Parameter	Description	Units
`E_0`	Reference internal energy.	J/mol
`V_0`	Reference volume.	m^3
`K_S`	Reference isentropic bulk modulus.	Pa
`Kprime_S`	First pressure derivative of the isentropic bulk modulus.	unitless
`Kprime_prime_S`	Second pressure derivative of the isentropic bulk modulus.	Pa^{-1}
`T_0`	Reference temperature on the principal isentrope.	K
`S_0`	Reference entropy on the principal isentrope.	J/K/mol
`n`	Number of atoms per formula unit.	unitless
`grueneisen_0`	Reference Gruneisen parameter on the principal isentrope.	unitless
`grueneisen_n`	Volume dependence of the Gruneisen parameter on the principal isentrope.	unitless
`grueneisen_prim`	First pressure derivative of the Gruneisen parameter on the principal isentrope.	Pa^{-1}
`theta`	Debye temperature on the principal isentrope.	K
`a`	Electronic heat capacity parameter A (length 2 array).	unitless
`b`	Electronic heat capacity parameter B (length 2 array).	unitless
`Theta`	Volume dependence parameters for electronic heat capacity (length 2 array).	unitless
`lmda`	Potential heat capacity parameter lambda (length 3 array).	unitless
`xi_0`	Reference xi parameter.	unitless
`F`	Reference F parameter (length 2 array).	unitless

Brosh CALPHAD EoS (Brosh, 2007) (brosh_calphad)

class burnman.eos.brosh_calphad.**BroshCalphad**

Bases: *EquationOfState*

Class for the high pressure CALPHAD equation of state by [BMS07].

This equation of state includes contributions from cold compression, quasiharmonic lattice vibrations, and thermal excitation of electrons.

Parameter	Description	Units
gibbs_coefficie	Coefficients for the piecewise polynomial fit to the Gibbs free energy at 1 bar. Format is a list of lists, where each sublist contains the maximum temperature for that segment and the coefficients in the following order: [const, T, T*ln(T), T^(-1), T^(-2), T^(-3), T^(-9), T^2, T^3, T^4, T^7, T^(1/2), ln(T)]	varies
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
n	Number of atoms per formula unit.	Dimensionless
a	Parameter array for cold compression (length 4 array).	Dimensionless
b	Parameter array for thermal excitation (length 2 array).	Dimensionless
c	[OPTIONAL] Parameter array for cold compression (length 4 array). If you do not know how to calculate it, do not add it to the parameters dictionary. It will be calculated from the other input parameters.	Dimensionless
delta	Parameter array for thermal excitation (length 2 array).	Dimensionless
theta_0	Reference Debye temperature.	K
grueneisen_0	Reference Grüneisen parameter.	Dimensionless

CORK EoS (Holland and Powell, 1991) (cork)

class burnman.eos.cork.**CORK**

Bases: *EquationOfState*

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

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

where $c_{i,j}$ are the CoRK parameters, T_c is the critical temperature, and P_c is the critical pressure.

Parameter	Description	Units
H_0	Reference enthalpy.	J/mol
S_0	Reference entropy.	J/(mol K)
Cp	Heat capacity parameters (length 4 array).	J/(mol K)
T_0	Reference temperature.	K
P_0	Reference pressure.	Pa
cork_params	CoRK parameters (length 4x2 array).	varies
cork_T	Critical temperature.	K
cork_P	Critical pressure.	Pa

De Koker and Stixrude Liquid EoS (de Koker and Stixrude, 2013) (dks_l)

class burnman.eos.dks_liquid.**DKS_L**

Bases: *EquationOfState*

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

This equation of state includes contributions from ideal gas (translational and electronic), electronic, and excess (bonding) components to the thermodynamic properties.

The parameters required for this equation of state do not all correspond to natural variables; please refer to [deKokerKarkiStixrude13] for details.

Parameter	Description	Units
V_0	Reference volume.	m^3
T_0	Reference temperature.	K
O_theta		
O_f		
m		
a		
zeta_0	Reference electronic strength parameter.	
xi	Power law exponent for the volume dependence on zeta.	Unitless
Tel_0	Reference temperature for electronic contributions.	K
eta	Power law exponent for the volume dependence on the electronic temperature.	Unitless
formula	Chemical formula of the substance.	Dict[str, float]
el_V_0	Reference volume for electronic contributions.	m^3
spin_a	Description of spin_a.	Length 2 list describing spin_a.
spin_b	Length 4 list describing spin_b.	Units of spin_b.

De Koker and Stixrude Solid EoS (de Koker and Stixrude, 2013) (dks_s)

class burnman.eos.dks_solid.**DKS_S**

Bases: *EquationOfState*

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

This equation of state is a somewhat modified version of that described in [SLB05].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
E_0	Reference internal energy.	J/mol
S_0	Reference entropy.	J/K/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of the reference bulk modulus.	unitless
grueneisen_0	Reference Grüneisen parameter.	unitless
q_0	Volume derivative of the Grüneisen parameter.	unitless
eta_s_0	Isotropic shear strain derivative of the Grüneisen parameter.	unitless
Cv	Constant volume heat capacity.	J/K/mol

Holland and Powell 1998 EoS (Holland and Powell, 1998) (hp98)

class burnman.eos.hp.**HP98**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
H_0	Enthalpy at the reference state.	J/mol
S_0	Entropy at the reference state.	J/(mol K)
Cp	Heat capacity coefficients at the reference pressure (length 4 list).	Various
V_0	Volume at the reference state.	m^3/mol
K_0	Isothermal bulk modulus at the reference state.	Pa
Kprime_0	Pressure derivative of bulk modulus at the reference state.	Dimensionless
a_0	Thermal expansivity at reference state.	1/K
dKdT_0	Temperature derivative of bulk modulus at reference state.	Pa/K

Holland and Powell Modified Tait Thermal EoS (Holland and Powell, 2011) (hp_tmt)

class burnman.eos.hp.**HP_TMT**

Bases: *EquationOfState*

Base class for the thermal equation of state based on the isothermal modified Tait equation of state (class MT), as described in [HollandPowell11].

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

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 thermal pressure is calculated using a Mie-Grüneisen-like function:

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

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

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

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
H_0	Enthalpy at the reference state.	J/mol
S_0	Entropy at the reference state.	J/(mol K)
Cp	Heat capacity coefficients at the reference pressure (length 4 list).	Various
V_0	Volume at the reference state.	m^3/mol
K_0	Isothermal bulk modulus at the reference state.	Pa
Kprime_0	Pressure derivative of bulk modulus at the reference state.	Dimensionless
Kdprime_0	Second pressure derivative of bulk modulus at the reference state.	Dimensionless
a_0	Thermal expansivity at reference state.	1/K
T_einstein	Einstein temperature.	K
n	Number of atoms in the unit formula.	Dimensionless

Holland and Powell Modified Tait Thermal Liquid EoS (Holland and Powell, 2011) (hp_tmtL)

class burnman.eos.hp.**HP_TMTL**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98], but with the Modified Tait as the static part, as described in [HollandPowell11].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
H_0	Enthalpy at the reference state.	J/mol
S_0	Entropy at the reference state.	J/(mol K)
Cp	Heat capacity coefficients at the reference pressure (length 4 list).	Various
V_0	Volume at the reference state.	m^3/mol
K_0	Isothermal bulk modulus at the reference state.	Pa
Kprime_0	Pressure derivative of bulk modulus at the reference state.	Dimensionless
Kdprime_0	Second pressure derivative of bulk modulus at the reference state.	Dimensionless
a_0	Thermal expansivity at reference state.	1/K
dKdT_0	Temperature derivative of bulk modulus at reference state.	Pa/K

Mie-Gruneisen-Debye 2nd-order EoS (Matas et al., 2007) (mgd2)

class burnman.eos.mie_grueneisen_debye.**MGD2**

Bases: *MGDBase*

MGD equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

Mie-Gruneisen-Debye 3rd-order EoS (Matas et al., 2007) (mgd3)

class burnman.eos.mie_grueneisen_debye.**MGD3**

Bases: *MGDBase*

Mie-Grueneisen-Debye equation of state with third order finite strain expansion for the shear modulus.

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 [SLB05]. 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.

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of the bulk modulus at reference pressure.	Dimensionless
`molar_mass`	Molar mass of the mineral.	kg/mol
`n`	Number of atoms per formula unit.	Dimensionless
`Debye_0`	Debye temperature at reference state.	K
`grueneisen_0`	Grüneisen parameter at reference state.	Dimensionless
`q_0`	Volume dependence of the Grüneisen parameter.	Dimensionless

Modular Mie-Gruneisen-Debye EoS (modular_mgd)

class burnman.eos.modular_mie_grueneisen_debye.**ModularMGD**

Bases: *EquationOfState*

This class implements a modular Mie-Grueneisen-Debye (MGD) equation of state (EoS) where the reference (isothermal) EoS and Debye temperature model can be chosen by the user.

The equation of state also supports electronic contributions to the Helmholtz energy and thermodynamic properties according to the model of [BK77].

Parameter	Description	Units
V_0	Reference volume	m^3
T_0	Reference temperature	K
bel_0	Electronic contribution parameter	unitless
gel	Electronic contribution exponent	unitless
n	Number of atoms per formula unit	atoms per formula unit
molar_mass	Molar mass	kg/mol
reference_eos	Reference isothermal equation of state. Must have methods for pressure, isothermal bulk modulus, and Gibbs energy.	*burnman.eos.EquationOfState*
debye_temperatu	Debye temperature model. An object providing the Debye temperature as a function of relative volume via value, and its first and second derivatives with respect to relative volume via dVrel and dVrel2.	callable. Examples include *burnman.eos.debye_temperature_models.SLB*, *burnman.eos.debye_temperature_models.PowerLawGammaSimple*, and *burnman.eos.debye_temperature_models.PowerLawGamma*.
anharmonic_ther	Anharmonic thermal model. An object providing the anharmonic contribution to the energy.	callable. Examples include *burnman.eos.anharmonic_thermal_models.Pade* and *burnman.eos.anharmonic_thermal_models.LogNormal*.

Modular Mie-Gruneisen-Debye EoS with Anharmonicity (modular_mgd_with_anharmonicity)

class burnman.eos.modular_mie_grueneisen_debye.**ModularMGDWithAnharmonicity**

Bases: *ModularMGD*

This class extends the ModularMGD class to include anharmonicity effects according to a simplifica-

tion of the model proposed by Wu and Wentzcovitch, 2009.

The basis of the anharmonic model is the definition of a scaled volume, V': $\ln(V'/V) == -c\ln(V/V_0)$. In this expression, V is the target volume, V_0 is a first order approximation to the volume at the same pressure and reference temperature T_0, and c is an anharmonicity parameter provided in the params dictionary as *c_anh*.

The anharmonic Helmholtz energy F is related to the scaled volume by the equation:

$$F(V,T) = F_h(V',T) + F_h(V,T_0) - F_h(V',T_0)$$

where F_h is the harmonic Helmholtz energy, potentially with electronic contributions.

Note: This model is not the same as that published in Wu and Wentzcovitch (2009). The results are expected to be similar, but the c parameter will in general need to be changed. This is because only a local approximation to the volume change between 0 K and the target temperature is used. This does not mean that the model is less able to capture the essential physics of the problem; indeed, the model of Wu and Wentzcovitch (2009) is only intended to be an effective ansatz.

Parameter	Description	Units
V_0	Reference volume	m^3
T_0	Reference temperature	K
bel_0	Electronic contribution parameter	unitless
gel	Electronic contribution exponent	unitless
n	Number of atoms per formula unit	atoms per formula unit
molar_mass	Molar mass	kg/mol
reference_eos	Reference isothermal equation of state. Must have methods for pressure, isothermal bulk modulus, and Gibbs energy.	*burnman.eos.EquationOfState*
debye_temperatu	Debye temperature model. An object providing the Debye temperature as a function of relative volume via value, and its first and second derivatives with respect to relative volume via dVrel and dVrel2.	callable. Examples include *burnman.eos.debye_temperature_models.SLB*, *burnman.eos.debye_temperature_models.PowerLawGammaSimple*, and *burnman.eos.debye_temperature_models.PowerLawGamma*.
anharmonic_ther	Anharmonic thermal model. An object providing the anharmonic contribution to the energy.	callable. Examples include *burnman.eos.anharmonic_thermal_models.Pade* and *burnman.eos.anharmonic_thermal_models.LogNormal*.
c_anh	Anharmonicity parameter	unitless

Stixrude and Lithgow-Bertelloni 2nd-order EoS (Stixrude and Lithgow-Bertelloni, 2005) (slb2)

class burnman.eos.slb.**SLB2**

Bases: *SLBBase*

SLB equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

Stixrude and Lithgow-Bertelloni 3rd-order EoS (Stixrude and Lithgow-Bertelloni, 2005) (slb3)

class burnman.eos.slb.**SLB3**

Bases: *SLBBase*

Third-order finite strain-Mie-Grueneisen-Debye equation of state detailed in [SLB05].

This equation of state combines the third-order Birch-Murnaghan isothermal equation of state with thermal corrections derived from the Mie-Grüneisen-Debye equation of state with the quasi-harmonic approximation:

$$\begin{aligned}
\mathcal{F}_{th}(T,\theta(V)) &= nRT\left(3\ln(1-e^{-\frac{\theta}{T}}) - \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau-1)}d\tau\right), \\
P_{th}(V,T) &= \frac{\gamma\Delta\mathcal{U}}{V}, \\
K_{th}(V,T) &= (\gamma+1-q)\frac{\gamma\Delta\mathcal{U}}{V} - \gamma^2\frac{\Delta(C_V T)}{V}, \\
G_{th}(V,T) &= -\frac{\eta_S\Delta\mathcal{U}}{V}.
\end{aligned}$$

The expression for the Debye temperature $\theta(V)$ as a function of volume is given by:

$$\begin{aligned}
\theta(V) &= \theta_0\sqrt{\frac{\nu^2}{\nu_0^2}}, \\
\frac{\nu^2}{\nu_0^2} &= 1 + a_{ii}^{(1)}f + \frac{1}{2}a_{iikk}^{(2)}f^2, \\
f &= \frac{1}{2}\left[\left(\frac{V_0}{V}\right)^{\frac{2}{3}} - 1\right].
\end{aligned}$$

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:

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

where 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.

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].

Parameter	Description	Units
`V_0`	Volume at P = 10^5 Pa , T = 300 K	$\mathrm{m}^3/\mathrm{mol}$
`K_0`	Isothermal bulk modulus at P = 10^5 Pa, T = 300 K	Pa
`Kprime_0`	Pressure derivative of K_0	unitless
`G_0`	Shear modulus at P = 10^5 Pa, T = 300 K	Pa
`Gprime_0`	Pressure derivative of G_0	unitless
`molar_mass`	mass per mole formula unit	kg mol^{-1}
`n`	number of atoms per formula unit	atoms/formula unit
`Debye_0`	Debye Temperature	K
`grueneisen_0`	Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
`q_0`	Volume dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
`eta_s_0`	Shear strain dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless

This equation of state is substantially the same as Mie-Gruneisen-Debye. 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.

Stixrude and Lithgow-Bertelloni 3rd-order Conductive EoS (Stixrude and Lithgow-Bertelloni, 2024) (slb3-conductive)

class burnman.eos.slb.**SLB3Conductive**

Bases: *SLBBase*

SLB equation of state with third order finite strain expansion for the shear modulus and a contribution to the Helmholtz free energy that arises from the thermal excitation of electrons [BK77].

For more details on the underlying EoS, we refer readers to [SLB05] and the documentation for *burnman.eos.slb.SLB3*. The electronic contribution to the Helmholtz free energy is given by:

$$\mathcal{F}_{el}(V,T) = -\frac{1}{2} b_0^{el} V \left(\frac{V}{V_0}\right)^{g^{el}} (T^2 - T_0^2)$$

Parameter	Description	Units
V_0	Volume at P = 10^5 Pa , T = 300 K	m^3/mol
K_0	Isothermal bulk modulus at P = 10^5 Pa, T = 300 K	Pa
Kprime_0	Pressure derivative of K_0	unitless
G_0	Shear modulus at P = 10^5 Pa, T = 300 K	Pa
Gprime_0	Pressure derivative of G_0	unitless
molar_mass	mass per mole formula unit	kg mol^{-1}
n	number of atoms per formula unit	atoms/formula unit
Debye_0	Debye Temperature	K
grueneisen_0	Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
q_0	Volume dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
eta_s_0	Shear strain dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
bel_0	Electronic bulk modulus coefficient	Pa
gel	Electronic bulk modulus volume exponent	unitless

3.2.4 Mineral Property Modifiers

Standard thermal equations of state typically consider the effects of strain and quasiharmonic lattice vibrations on the 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 energy of a phase, which are not typically included in standard equations of state. Corrections for these additional processes can, to first order, be applied as additive modifiers, ignoring any explicit coupling with the equation of state.

Burnman currently includes implementations of the following:

- *Landau tricritical model (Putnis, 1992) (available as landau)*
- *Landau tricritical model (Stixrude and Lithgow-Bertelloni, 2022) (available as landau_slb_2022)*

- *Landau tricritical model (Holland and Powell, 1998) (available as landau_hp)*
- *Linear in P and T (available as linear)*
- *Bragg-Williams model (available as bragg_williams)*
- *Magnetic ordering (Chin et al., 1987) (available as magnetic_chs)*
- *Debye model excess (Helmholtz) (available as debye)*
- *Debye model excess (internal energy) (available as debye_delta)*
- *Einstein model excess (Helmholtz) (available as einstein)*
- *Einstein model excess (internal energy) (available as einstein_delta)*

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

$$\begin{aligned}
\mathcal{G} &= \mathcal{G}_\mathrm{o} + \mathcal{G}_\mathrm{m}, \\
S &= S_\mathrm{o} - \left(\frac{\partial \mathcal{G}}{\partial T}\right)_\mathrm{m}, \\
V &= V_\mathrm{o} + \left(\frac{\partial \mathcal{G}}{\partial P}\right)_\mathrm{m}, \\
K_T &= V/\left(\left(\frac{V_\mathrm{o}}{K_{T\mathrm{o}}}\right) - \left(\frac{\partial^2 \mathcal{G}}{\partial P^2}\right)_\mathrm{m}\right), \\
C_p &= C_{p\mathrm{o}} - T\left(\frac{\partial^2 \mathcal{G}}{\partial T^2}\right)_\mathrm{m}, \\
\alpha &= \left(\alpha_\mathrm{o} V_\mathrm{o} + \left(\frac{\partial^2 \mathcal{G}}{\partial P \partial T}\right)_\mathrm{m}\right)/V
\end{aligned}$$

where the subscript "o" refers to properties calculated from the underlying equation of state, and the subscript "m" refers to contributions from the property modifier. Other derived properties are then calculated from these modified thermodynamic properties using standard thermodynamic relationships:

$$\begin{aligned}
\mathcal{H} &= \mathcal{G} + TS, \\
\mathcal{F} &= \mathcal{G} - PV, \\
C_v &= C_p - VT\alpha^2 K_T, \\
\gamma &= \frac{\alpha K_T V}{C_v}, \\
K_S &= K_T \frac{C_p}{C_v}
\end{aligned}$$

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}],
                          ["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.

3.2.4.1 Landau tricritical model (Putnis, 1992) (available as landau)

burnman.eos.property_modifiers.**landau_excesses**(*pressure*, *temperature*, *params*)

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.

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:

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

If temperature is below the critical temperature, Q is between 0 and 1. The Gibbs energy can be

described thus:

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

Table 1: Parameters

Parameter	Description
`Tc_0`	Critical temperature at reference pressure (K)
`S_D`	Entropy parameter (J/mol/K)
`V_D`	Volume parameter (m^3/mol)

3.2.4.2 Landau tricritical model (Stixrude and Lithgow-Bertelloni, 2022) (available as landau_slb_2022)

`burnman.eos.property_modifiers.`**`landau_slb_2022_excesses`**(*pressure*, *temperature*, *params*)

Applies a tricritical Landau correction to the properties of an endmember which undergoes a displacive phase transition. This correction follows [SLB22], and is done relative to the state with order parameter $Q = 1$.

The order parameter of this formulation can exceed one, at odds with [Putnis92], but in better agreement with atomic intuition [SLB22]. Nevertheless, this implementation is still not perfect, as the excess entropy (and heat capacity) terms are not equal to zero at 0 K. Q is limited to values less than or equal to 2 to avoid unrealistic stabilisation at ultrahigh pressure.

Table 2: Parameters

Parameter	Description
`Tc_0`	Critical temperature at reference pressure (K)
`S_D`	Entropy parameter (J/mol/K)
`V_D`	Volume parameter (m^3/mol)

3.2.4.3 Landau tricritical model (Holland and Powell, 1998) (available as landau_hp)

burnman.eos.property_modifiers.**landau_hp_excesses**(*pressure*, *temperature*, *params*)

Applies a tricritical Landau correction similar to that described above. However, this implementation follows [HollandPowell11], who compute properties relative to the standard state.

Includes the correction published within landaunote.pdf (Holland, pers. comm), which 'corrects' the terms involving the critical temperature Tc / Tc*.

Note that this implementation allows the order parameter Q to be greater than one.

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

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

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

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

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

$$\begin{aligned}
\frac{\partial^2 \mathcal{G}}{\partial P^2} = -\frac{V_D^2}{2S_D Tc_0 Q^2}, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} = -\frac{S_D}{2Tc_0 Q^2}, \\
\frac{\partial^2 \mathcal{G}}{\partial P \partial T} = \frac{V_D}{2Tc_0 Q^2}
\end{aligned}$$

Table 3: Parameters

Parameter	Description
P_0	Reference pressure (Pa)
T_0	Reference temperature (K)
Tc_0	Critical temperature at reference pressure (K)
S_D	Entropy parameter (J/mol/K)
V_D	Volume parameter (m^3/mol)

3.2.4.4 Linear in P and T (available as linear)

burnman.eos.property_modifiers.**linear_excesses**(*pressure*, *temperature*, *params*)

A simple linear correction in pressure and temperature.

The excess Gibbs energy and its derivatives are given by:

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

This form of excess is extremely useful as a first order tweak to free energies (especially in solid solutions where data on all endmembers may not be available).

Table 4: Parameters

Parameter	Description
delta_E	Energy correction (J/mol)
delta_S	Entropy correction (J/mol/K)
delta_V	Volume correction (m^3/mol)

3.2.4.5 Bragg-Williams model (available as bragg_williams)

burnman.eos.property_modifiers.**bragg_williams_excesses**(*pressure*, *temperature*, *params*)

The Bragg-Williams model is a symmetric solution model with an excess configurational entropy term determined by the specifics of order-disorder in the mineral, multiplied by some empirical factor. Expressions for the excess Gibbs free energy can be found in [HP96].

Excess properties assume that order-disorder processes are rapid compared with the timescales of pressure and temperature changes (i.e., equilibrium is maintained).

This may not be reasonable for order-disorder, especially for slow or coupled diffusers (Si-Al, for example). Properties for minerals that order-disorder slowly should be calculated using an explicit solid solution model.

Table 5: Parameters

Parameter	Description
deltaH	Enthalpy change (J/mol)
deltaV	Volume change (m^3/mol)
Wh	Enthalpy interaction parameter (J/mol)
Wv	Volume interaction parameter (m^3/mol)
n	Related to the number of available sites for ordering
factor	An empirical factor

3.2.4.6 Magnetic ordering (Chin et al., 1987) (available as magnetic_chs)

burnman.eos.property_modifiers.**magnetic_excesses_chs**(*pressure*, *temperature*, *params*)

This model approximates the excess energy due to magnetic ordering. It was originally described in [CHS87]. The expressions used in this implementation can be found in [Sun91].

Table 6: Parameters

Parameter	Description
structural_paramet	A dimensionless parameter related to the crystal structure
curie_temperature	A list of length 2: zero pressure Curie temperature (K) and pressure dependence (K/Pa)
magnetic_moment	A list of length 2: zero pressure magnetic moment (Bohr magnetons) and pressure dependence (Bohr magnetons/Pa)

3.2.4.7 Debye model excess (Helmholtz) (available as debye)

burnman.eos.property_modifiers.**debye_excesses**(*pressure*, *temperature*, *params*)

Applies an excess contribution based on a Debye model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= F_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -S_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{C_{V,\text{Debye}}}{T}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess heat capacity tends toward a constant value at high temperature.

Table 7: Parameters

Parameter	Description
`Cv_inf`	Heat capacity at infinite temperature (J/mol/K).
`Theta_0`	Debye temperature (K).

3.2.4.8 Debye model excess (internal energy) (available as debye_delta)

`burnman.eos.property_modifiers.`**`debye_delta_excesses`**(*pressure*, *temperature*, *params*)

Applies an excess contribution based on a Debye model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= -U_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -C_{V,\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{dC_{V,\text{Debye}}}{dT}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess entropy tends toward a constant value at high temperature and behaves like the heat capacity of a Debye model at finite temperature.

Table 8: Parameters

Parameter	Description
`S_inf`	Entropy at infinite temperature (J/mol/K).
`Theta_0`	Einstein temperature (K).

3.2.4.9 Einstein model excess (Helmholtz) (available as einstein)

`burnman.eos.property_modifiers.`**`einstein_excesses`**(*pressure*, *temperature*, *params*)

Applies an excess contribution based an Einstein model. The excess Gibbs energy and its derivatives

are given by:

$$\begin{aligned}
\mathcal{G} &= F_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -S_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{C_{V,\text{Einstein}}}{T}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess heat capacity tends toward a constant value at high temperature.

Table 9: Parameters

Parameter	Description
`Cv_inf`	Heat capacity at infinite temperature (J/mol/K).
`Theta_0`	Einstein temperature (K).

3.2.4.10 Einstein model excess (internal energy) (available as einstein_delta)

`burnman.eos.property_modifiers.`**`einstein_delta_excesses`**(*pressure*, *temperature*, *params*)

Applies an excess contribution based an Einstein model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= -U_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -C_{V,\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{dC_{V,\text{Einstein}}}{dT}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess entropy tends toward a constant value at high temperature and behaves like the heat capacity of an Einstein model at finite temperature.

Table 10: Parameters

Parameter	Description
`S_inf`	Entropy at infinite temperature (J/mol/K).
`Theta_0`	Einstein temperature (K).

3.3 Solutions

Many phases (whether minerals, melts, fluids or gases) can exist over a finite region of composition space. These spaces are bounded by endmembers (which may themselves not be stable), and each phase can then be described as a solution of those endmembers.

In a mechanically and thermally equilibrated mechanical mixture (i.e., not a solution), the Gibbs energy of the mixture is simply the sum of the Gibbs energies of the individual components. However, in a solution, the Gibbs energy also includes an entropic contribution from the mixing of the different components. This entropic contribution generally lowers the overall Gibbs energy of the phase, making the solution more stable than a mechanical mixture of its endmembers. In addition, there may be enthalpic contributions to the Gibbs energy of mixing, which can either further stabilize or destabilize the solution relative to the mechanical mixture, such that:

$$\mathcal{G}_{\text{solution}} = \sum_i n_i \mathcal{G}_i + \Delta \mathcal{G}_{\text{mixing}}$$

$$\Delta \mathcal{G}_{\text{mixing}} = \Delta \mathcal{H}_{\text{mixing}} - T \Delta S_{\text{mixing}}$$

where n_i is the number of moles of component i, $\mathcal{G}_i$ is the molar Gibbs energy of component i, $\Delta \mathcal{H}_{\text{mixing}}$ is the enthalpy of mixing, and ΔS_{mixing} is the entropy of mixing.

It is important to recognise that the components of a solution must be compatible with the structure of the phase; that is, they must all be contained within the same crystallographic framework. They do not need to have unique compositions. For example, we can model a displacive phase transition through a solution of two identical components which differ only in the element of symmetry lost during the transition, and we can model order-disorder transitions through solutions of components which differ only in the distribution of cations over crystallographic sites.

Derivatives of the Gibbs energy of the solution can now be defined with respect to amounts of components, in addition to pressure and temperature. The first derivatives give rise to so-called chemical potentials, which are extremely useful in defining phase equilibria.

In BurnMan, we define *burnman.Solution* to represent a solution phase. Each instance of this class contains a *burnman.SolutionModel* which defines how the Gibbs energy of the solution is calculated from its components. The following sections describe these classes in more detail.

3.3.1 The Solution Class

class burnman.**Solution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

The Solution class has overall responsibility for calculating thermodynamically self-consistent properties of a solution phase.

In this class, solutions are treated as a collection of independent endmember phases (components), plus a model that describes the excess properties of the solution. It stores the current state (pressure, temperature and molar fractions of all the endmembers) of the solution, and interrogates a SolutionModel object to calculate the excess properties of the solution. It then combines these with the properties of the individual endmember phases (also stored in the SolutionModel) to calculate the overall properties

of the solution. This is done using standard thermodynamic relations:

$$G = \sum_i X_i G_i + G^{\text{excess}}$$
$$V = \sum_i X_i V_i + V^{\text{excess}}$$
$$S = \sum_i X_i S_i + S^{\text{excess}}$$
$$C_p = \sum_i X_i C_{p,i} + C_p^{\text{excess}}$$
$$\alpha V = \sum_i X_i \alpha_i V_i + (\alpha V)^{\text{excess}}$$
$$\frac{V}{K_T} = \sum_i X_i \frac{V_i}{K_{T,i}} + \left(\frac{V}{K_T}\right)^{\text{excess}}$$

After instantiation, the user can set the composition of the solution using the set_composition() method, and the pressure and temperature using the set_state() method.

Instantiation of a Solution object can be achieved in just a few lines:

```
from burnman import Solution, SolutionModel

# Create a SolutionModel object (explained in the documentation for that␣
↪class)
solution_model = SolutionModel(...)

# Create a Solution object
solution = Solution(name="MySolution", solution_model=solution_model, molar_
↪fractions=...)
```

where the arguments are as follows:

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.SolutionModel*) – The SolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The initial molar fractions of each end-member in the solution. Can be reset using the set_composition() method.

3.3.2 Solution Model Classes

3.3.2.1 Mean-field solution models

In BurnMan, all solution models are based on mean-field theories of solutions. These theories, introduced by Bragg and Williams in 1934, assume that each component in the solution experiences an average environment created by all other components, rather than accounting for specific local interactions between different species. This simplification allows for tractable mathematical descriptions of solution behaviour.

Each of the solution models below takes an `endmembers` argument, which is a list of lists. Each sub-list contains two items. The first is an instance of *burnman.Mineral* representing an independent endmember (or component) of the solution. The second is a string, which represents the occupancies of different species on distinct sites in the solution. For example, low pressure silicate garnets have two distinct sites on which mixing takes place; a dodecahedral site (of which there are three per unit cell on an eight-cation basis) and octahedral site (of which there are two per unit cell). A third tetrahedral cation site (three per unit cell) is usually assumed to be occupied solely by silicon, and therefore can be ignored in solution calculations. The chemical formula of many low pressure garnets exist within the solution:

$$[\mathrm{Mg,Fe,Mn,Ca}]_3[\mathrm{Al,Fe^{3+},Cr}]_2\mathrm{Si}_3\mathrm{O}_{12}$$

where the dodecahedral site is occupied by Mg, Fe, Mn and Ca, and the octahedral site is occupied by Al, Fe and Cr. The pyrope endmember, composition $\mathrm{Mg}_3\mathrm{Al}_2\mathrm{Si}_3\mathrm{O}_{12}$, would have the corrsponding site occupancy string `[Mg]3[Al]2Si3O12`, while andradite would have `[Ca]3[Fef]2Si3O12`. BurnMan also allows for partial occupancies on sites; for example, disordered grandite with composition $\mathrm{Ca}_3\mathrm{AlFeSi}_3\mathrm{O}_{12}$ would have the site occupancy string `[Ca]3[Al0.5Fef0.5]2Si3O12`. Note that distinct species do not need to be named as elements; the only requirement is that the names start with a capital letter and are followed by lower-case letters. Atom proportions and site multiplicities can be expressed as decimal numbers or fractions.

In BurnMan, the multiplicities of each site are allowed to vary linearly between endmembers. This is known as a Temkin model [Tem45], and is useful for modelling phases where the site multiplicities are not fixed by crystal structure, such as glasses or melts.

3.3.2.2 Implemented solution models

MechanicalSolution

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

Bases: *SolutionModel*

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of mixing; the total gibbs free energy of the solution is equal to the dot product of the molar gibbs free energies and molar fractions of the constituent materials.

IdealSolution

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

Bases: *SolutionModel*

A solution model representing ideal mixing between endmembers. In ideal solutions, the enthalpy of mixing is equal to zero; i.e. chemical mixing of endmembers at fixed pressure and temperature does not involve the absorption or release of heat. The only contribution to the Gibbs energy of mixing is the configurational entropy S_{conf}:

$$\mathcal{G}_{\mathrm{mixing}} = -T\Delta S_{\mathrm{conf}},$$
$$S_{\mathrm{conf}} = Rx_c^s \ln \frac{x_c^s}{\sum_{c^s} x_c^s}$$

where s is a site in the lattice, c are the species mixing on site s. x_c^s is the absolute number of species c on site s in the lattice; it is calculated by multiplying the proportion of the species on the site by the

multiplicity of the site per formula unit and the number of moles of formula units. This class allows the site multiplicities to vary linearly between endmembers. This is known as a Temkin model [Tem45].

AsymmetricRegularSolution

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

Bases: *IdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Gibbs energy is given by the expression [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.

Parameters

- **endmembers** (*list of tuples*) – A list of tuples defining the endmembers of the solution. Each tuple contains a *burnman.Mineral* object and a string defining the site formula of that endmember.
- **alphas** (*list of floats*) – A list of van Laar parameters governing asymmetry in the excess properties.
- **energy_interaction** (*list of lists of floats*) – A 2D list of interaction parameters defining the energetic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **volume_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the volumetric interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **entropy_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the entropic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.

SymmetricRegularSolution

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

Bases: *AsymmetricRegularSolution*

Solution model implementing the Symmetric Regular Solution Model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class where all van Laar parameters are equal to unity.

The excess terms in the Symmetric Regular Solution Model 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.

Parameters

- **endmembers** (*list of tuples*) – A list of tuples defining the endmembers of the solution. Each tuple contains a *burnman.Mineral* object and a string defining the site formula of that endmember.
- **energy_interaction** (*list of lists of floats*) – A 2D list of interaction parameters defining the energetic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **volume_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the volumetric interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **entropy_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the entropic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.

SubregularSolution

class burnman.classes.solutionmodel.**SubregularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *volume_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *IdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess nonconfigurational Gibbs energy is given by the expression:

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

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **volume_interaction** (*list of list of lists*) – The binary endmember interaction volumes. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **volume_ternary_terms** (*list of lists*) – The ternary interaction volumes. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

FunctionSolution

class burnman.classes.solutionmodel.**FunctionSolution**(*endmembers*, *excess_gibbs_function*)

Bases: *IdealSolution*

Solution model implementing a generalized solution model. The extensive excess nonconfigurational Gibbs energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Gibbs energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.

- **excess_gibbs_function** (*function*) – The nonconfigurational Gibbs energy function with arguments pressure, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Gibbs energy must also double.

PolynomialSolution

class burnman.classes.solutionmodel.**PolynomialSolution**(*endmembers*, *ESV_interactions=None*, *interaction_endmembers=[]*, *endmember_coefficients_and_interactions=None*, *transformation_matrix=None*)

Bases: *IdealSolution*

Solution model implementing a general polynomial solution model.

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.
- **ESV_interactions** (*list of lists*) – A list containing lists where the first three elements are energy, entropy and volume interactions and the rest of the elements are indices of the transformed endmembers to which those interactions correspond, and the proportion exponents. For example, [2., 0., 0., 0, 4, 1, 1] would correspond to an interaction of 2*p'[0]^4*p'[1]^1.
- **interaction_endmembers** (list of *burnman.Mineral* objects) – A list of minerals involved in the interaction terms.
- **endmember_coefficients_and_interactions** (*list of lists*) – list of lists A list containing lists where the first n elements are coefficients for each of the interaction_endmembers and the rest of the elements are indices of the transformed endmembers to which those interactions correspond. For example, [1., 0., -1., 0, 4, 1, 1] would correspond to an interaction of (mbr[0].gibbs - mbr[2].gibbs)*p'[0]^4*p'[1]^1.
- **transformation_matrix** (*2D numpy array*) – The interactions for a given solution may be most compactly expressed not as a polynomial function of the proportions of the endmembers, but a polynomial function of a linearly transformed set. This parameter is a square numpy array A, where p'i = A_ij p_j

3.4 Elastic Solution Class

Elastic solution models in BurnMan are much like standard *Solution Model Classes*. The only difference is that instead of considering excess **Gibbs energy** of mixing when all of the endmembers are evaluated at the same **pressure** and temperature, elastic solution models consider excess **Helmholtz energy** of mixing when all of the endmembers are evaluated at the same **volume** and temperature.

As a consequence of the different thermodynamic potential being used, elastic solution models can include **excess contributions to pressure**, rather than **excess contributions to volume**.

In BurnMan, we define *burnman.ElasticSolution* to represent a solution phase. Each instance of this class contains a *burnman.ElasticSolutionModel* which defines how the Helmholtz energy of the solution is calculated from its components. The following sections describe these classes in more detail.

3.4.1 The Elastic Solution Class

class burnman.**ElasticSolution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all Elastic solutions. Site occupancies, endmember activities and the constant and volume and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state() and set_composition.

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

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and V derivatives in J/K/mol and Pa/mol.

The parameters are relevant to all Elastic solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.ElasticSolutionModel*) – The ElasticSolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

3.4.2 Elastic Solution Model Classes

3.4.2.1 Mean-field elastic solution models

Just like the *burnman.SolutionModel* implementations, *burnman.ElasticSolutionModel* classes in BurnMan are based on mean-field theories of solutions. For more details on these theories, and how they relate to BurnMan solution objects, please refer to *Solution Model Classes*.

3.4.2.2 Implemented elastic solution models

ElasticMechanicalSolution

class burnman.classes.elasticsolutionmodel.**ElasticMechanicalSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of

mixing; the total Helmholtz energy of the solution is equal to the dot product of the molar Helmholtz energies and molar fractions of the constituent materials.

ElasticIdealSolution

class burnman.classes.elasticsolutionmodel.**ElasticIdealSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

A class representing an ideal solution model. Calculates the excess Helmholtz energy and entropy due to configurational entropy. Excess internal energy and volume are equal to zero.

The multiplicity of each type of site in the structure is allowed to change linearly as a function of endmember proportions. This class is therefore equivalent to the entropic part of a Temkin-type model [Tem45].

ElasticAsymmetricRegularSolution

class burnman.classes.elasticsolutionmodel.**ElasticAsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*

Bases: *ElasticIdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Helmholtz energy is given by the expression:

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

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

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

ElasticSymmetricRegularSolution

class burnman.classes.elasticsolutionmodel.**ElasticSymmetricRegularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*)

Bases: *ElasticAsymmetricRegularSolution*

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

ElasticSubregularSolution

class burnman.classes.elasticsolutionmodel.**ElasticSubregularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *pressure_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *ElasticIdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess conconfigurational Helmholtz energy is given by the expression:

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

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **pressure_interaction** (*list of list of lists*) – The binary endmember interaction pressures. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **pressure_ternary_terms** (*list of lists*) – The ternary interaction pressures. Each list should contain four entries: the indices i, j, k and the value of the interaction.

- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

ElasticFunctionSolution

class burnman.classes.elasticsolutionmodel.**ElasticFunctionSolution**(*endmembers*, *excess_helmholtz_function*)

Bases: *ElasticIdealSolution*

Solution model implementing a generalized elastic solution model. The extensive excess nonconfigurational Helmholtz energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Helmholtz energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **excess_helmholtz_function** (*function*) – The nonconfigurational Helmholtz energy function with arguments volume, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Helmholtz energy must also double.

3.5 Composite Class

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

Bases: *Material*

The composite class in BurnMan is designed to calculate the properties of multi-phase assemblages. Each phase within the composite is represented by an instance of the Material class (typically a Mineral or Solution). The composite class calculates the bulk properties of the assemblage based on the properties and proportions of the individual phases.

All of the **thermodynamic** properties (Gibbs free energy, enthalpy, entropy, volume, heat capacity etc.) are calculated by assuming that any changes in pressure or temperature of the composite as a whole are reflected equally in all of the individual phases, but that no reactions occur either within or between phases. For a composite that calculates properties assuming some or all reactions between phases, can take place ion the timescales of changes in pressure and temperature, see *Relaxed Composite Class*.

The **elastic** properties (bulk and shear moduli, seismic velocities etc.) of a composite material on short timescales are often not well-approximated by assuming homogeneous changes in pressure and temperature. This is because differential stresses can develop between phases due to their differing elastic and thermal properties. To account for this, the composite class allows the user to specify an averaging scheme for the elastic moduli of the composite. The default averaging scheme is Voigt-Reuss-Hill, which is appropriate for many applications. Other *Averaging Schemes* are available.

Example:

```
from burnman import Composite
from burnman.minerals import SLB2011

# Create a composite made of 70% olivine and 30% orthopyroxene
phases = [SLB2011.mg_fe_olivine(), SLB2011.orthopyroxene()]
fractions = [0.7, 0.3]  # molar fractions
composite = Composite(phases, fractions, fraction_type='molar', name=
↪'Mantle composite')

# Set the compositions of the solution phases
# Endmembers of olivine are forsterite and fayalite
# Endmembers of orthopyroxene are enstatite, ferrosilite, Mg-tschermaks and␣
↪Orthodiopside
composite.phases[0].set_composition([0.9, 0.1])  # Fo90
composite.phases[1].set_composition([0.8, 0.2, 0.0, 0.0])  # En80

# Set the state of the composite (1 GPa, 1500 K)
composite.set_state(1e9, 1500)

# Change the fractions of the phases
composite.set_fractions([0.6, 0.4])

# Change the averaging scheme to Hashin-Shtrikman lower bound
composite.set_averaging_scheme("HashinShtrikmanLower")
```

Parameters

- **phases** (list of *burnman.Material*) – List of phases.
- **fractions** (*list of floats*) – molar or mass fraction for each phase.
- **fraction_type** (*str*) – 'molar' or 'mass' (optional, 'molar' as standard) specify whether molar or mass fractions are specified.
- **name** (*str*) – Name of the composite (optional).

3.6 PerpleX Class

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

Bases: *Material*

PerpleX material class. This class creates a material based on thermodynamic and thermoelastic data read from a 2D PerpleX tab file. This file should be in the standard format output by the werami program, 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 can only be queried after setting the pressure and temperature using set_state(). These properties 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.

Parameters

- **tab_file** (*str*) – Path to the 2D PerpleX tab file.
- **name** (*str*) – Name of the material.

CHAPTER

FOUR

ANISOTROPIC MATERIALS

In addition to BurnMan's standard materials, which have properties that are a function of pressure, temperature, and composition, BurnMan also includes classes for anisotropic elastic materials. These materials have elastic properties that depend on stress rather than pressure, or strain rather than volumetric compression.

Because most natural materials in the Earth can only withstand small deviatoric stresses before yielding or fracturing, the anisotropic materials in BurnMan have properties that are defined as a function of pressure, but these properties include those that describe the response of the material to small deviatoric stresses. Such properties include the full elastic tensor (isothermal or isentropic), the thermal expansion tensor, and the heat capacity at constant strain.

4.1 Anisotropic Materials

4.1.1 Anisotropic Material Base Class

class burnman.classes.anisotropy.**AnisotropicMaterial**(*rho*, *cijs*)

Bases: *Material*

A base class for anisotropic elastic materials. The base class is initialised with a density and a full isentropic stiffness tensor in Voigt notation. It can then be interrogated to find the values of different properties, such as bounds on seismic velocities. There are also several functions which can be called to calculate properties along directions oriented with respect to the isentropic elastic tensor.

See [MHS11] and https://docs.materialsproject.org/methodology/elasticity/ for mathematical descriptions of each function.

property density

Returns the density of this material.

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

Returns

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

Return type

float

property isentropic_bulk_modulus_voigt

Returns
The Voigt bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_reuss

Returns
The Reuss bound on the isentropic bulk modulus [Pa].

Return type
float

property isentropic_bulk_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_shear_modulus_voigt

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_reuss

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the lateral response to loading.

Return type
float

christoffel_tensor(*propagation_direction*)

Calculates the adiabatic Christoffel tensor.

The isentropic stiffness tensors provided in this and other anisotropic classes correspond to the isentropic derivative of stress with respect to symmetric infinitesimal strain. This corresponds to the isentropic elastic tensor B of Wallace (1967; doi:10.1103/PhysRev.162.776).

The Christoffel (acoustic) matrix is the contraction T_ik = S_ijkl n_j n_l, where S is Wallace's isentropic equation-of-motion tensor (not to be confused with the isentropic compliance tensor). For a unit direction n, contracting S gives T = sym(T_B) + ((n . sigma . n) I - sigma) / 2, where T_B_ik = B_ijkl n_j n_l and sigma is the tension-positive Cauchy stress. This derivation follows from Wallace (1967), using his Equations (2.24), (2.28), and (2.36).

Parameters
propagation_direction – Direction or array of directions with shape (..., 3), expressed in the same frame as the stress and stiffness tensors. Each direction is normalized before contraction.

Returns
Christoffel tensor in Pa, with shape (..., 3, 3).

Return type
numpy.ndarray

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction relative to the stiffness tensor [1/Pa].

Return type
float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction relative to the stiffness tensor [Pa].

Return type
float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor [Pa].

Return type
float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response directions relative to the stiffness tensor [unitless].

Return type
float

wave_velocities(*propagation_direction*)

Returns
The compressional wave velocity, and two shear wave velocities in a given propagation direction [m/s].

Return type
list, containing the wave speeds and directions of particle motion relative to the stiffness tensor

4.1.2 Derived classes

class burnman.classes.anisotropy.**IsotropicMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C12, C44] (i.e. lambda and mu, the Lame parameters)

class burnman.classes.anisotropy.**CubicMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C44]

class burnman.classes.anisotropy.**HexagonalMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C13, C33, C44]

class burnman.classes.anisotropy.**TetragonalMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C13, C33, C44, C66] or [C11, C12, C13, C16, C33, C44, C66]

class burnman.classes.anisotropy.**RhombohedralMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C13, C14, C33, C44, C66] or [C11, C12, C13, C14, C15, C33, C44, C66]

class burnman.classes.anisotropy.**OrthorhombicMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C13, C22, C23, C33, C44, C55, C66]

class burnman.classes.anisotropy.**MonoclinicMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [C11, C12, C13, C15, C22, C23, C25, C33, C35, C44, C46, C55, C66]

class burnman.classes.anisotropy.**TriclinicMaterial**(*rho*, *cijs*)

Bases: *AnisotropicMaterial*

A class derived from the AnisotropicMaterial base class Initialization takes two input parameters; rho and [Cij, where 1<=i<=6 and i<=j<=6]

4.2 Anisotropic Mineral Class

class burnman.classes.anisotropicmineral.**AnisotropicMineral**(*isotropic_mineral*, *cell_parameters*, *anisotropic_parameters*, *frame_convention=None*, *psi_function=None*, *orthotropic=None*)

Bases: *Mineral*, *AnisotropicMaterial*

A class implementing the anisotropic mineral equation of state described in [Myh22]. This class is derived from both Mineral and AnisotropicMaterial, and inherits most of the methods from these classes.

Instantiation of an AnisotropicMineral takes three required arguments; a reference Mineral (i.e. a standard isotropic mineral which provides volume as a function of pressure and temperature), cell_parameters, which give the lengths of the molar cell vectors and the angles between them (see *cell_parameters_to_vectors()*), and an anisotropic parameters object, which should be either a 4D array of anisotropic parameters or a dictionary of parameters which describe the anisotropic behaviour of the mineral. For a description of the physical meaning of the parameters in the 4D array, please refer to the code or to the original paper.

For non-orthotropic materials, the argument frame_convention should be set to define the orientation of the reference frame relative to the crystallographic axes (see *cell_parameters_to_vectors()*).

If the user chooses to define their parameters as a dictionary, they must also provide a function to the psi_function argument that describes how to compute the tensors Psi, dPsidf and dPsidPth (in Voigt

form). The function arguments should be f, Pth and params, in that order. The output variables Psi, dPsidf and dPsidth must be returned in that order in a tuple. The user should also explicitly state whether the material is orthotropic or not by supplying a boolean to the orthotropic argument.

States of the mineral can only be queried after setting the pressure and temperature using set_state().

This class is available as `burnman.AnisotropicMineral`.

All the material parameters are expected to be in plain SI units. This means that the elastic moduli should be in Pascals and NOT Gigapascals. Additionally, the cell parameters should be in m/(mol formula unit) and not in unit cell lengths. To convert unit cell lengths given in Angstrom to molar cell parameters you should multiply by 10^(-10) * (N_a / Z)^1/3, where N_a is Avogadro's number and Z is the number of formula units per unit cell. You can look up Z in many places, including www.mindat.org.

Finally, it is assumed that the unit cell of the anisotropic material is aligned in a particular way relative to the coordinate axes (the anisotropic_parameters are defined relative to the coordinate axes). The crystallographic a-axis is assumed to be parallel to the first spatial coordinate axis, and the crystallographic b-axis is assumed to be perpendicular to the third spatial coordinate axis.

4.3 Anisotropic Solution Class

class burnman.classes.anisotropicsolution.**AnisotropicSolution**(*name*, *solution_model*, *psi_excess_function*, *anisotropic_parameters*, *molar_fractions=None*)

Bases: *Solution*, *AnisotropicMineral*

A class implementing the anisotropic solution model described in [Myh24]. This class is derived from Solution and AnisotropicMineral, and inherits most of the methods from those classes.

Instantiation of an AnisotropicSolution is similar to that of a scalar Solution, except that each of the endmembers must be an instance of an AnisotropicMineral, and an additional function and parameter dictionary are passed to the constructor of AnisotropicSolution that describe excess contributions to the anisotropic state tensor (Psi_xs) and its derivatives with respect to volume and temperature. The function arguments should be ln(V), Pth, p (a vector of proportions) and the parameter dictionary, in that order. The output variables Psi_xs_Voigt, dPsidf_Pth_Voigt_xs and dPsidPth_f_Voigt_xs (all 6x6 matrices) and dPsiIdp_xs (a 3 x 3 x n_endmember matrix) must be returned in that order in a tuple. States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.AnisotropicSolution`.

4.4 Hyperelastic Mineral Class

The `HyperelasticMineral` class extends the `AnisotropicMineral` class from hydrostatic states to finite, non-hydrostatic elastic deformations. Its independent variables are the cell vectors and temperature. Derivatives of the Helmholtz free energy with respect to those variables determines the stress, entropy, and other thermodynamic and thermoelastic properties. The underlying AnisotropicMineral supplies both the reference cell and the hydrostatic properties at each volume and temperature.

The following description of the `HyperelasticMineral` class uses fixed composition, tension-positive Cauchy stress $\boldsymbol{\sigma}$, and compression-positive pressure. Helmholtz energy $\mathcal{F}$, entropy S and heat capacities are **molar** quantities; V is the molar volume. Stresses and stiffnesses have units of Pa. Repeated Cartesian indices are summed, and : denotes contraction over two indices. Superscripts T and S on elastic tensors mean isothermal and isentropic, respectively. Fourth-order tensors use blackboard-bold symbols, such as $\mathbb{B}$, $\mathbb{C}$ and $\mathbb{S}$.

4.4.1 Cell geometry and the hydrostatic reference

Let $\mathbf{M}$ have the three molar cell vectors as its **columns**. BurnMan stores these as **rows** in the `cell_vectors` attribute, so that `M = mineral.cell_vectors.T`. For a positively oriented cell,

$$V = \det \mathbf{M}$$

where $\mathbf{M}$ is the cell tensor. At constant volume, only $\det \mathbf{M}$ is fixed; the individual cell lengths and angles are able to vary. When the state of the `HyperelasticMineral` is set with `set_state_with_molar_cell_vectors`, the cell vectors are stored in `cell_vectors` and the molar volume is computed. The underlying `AnisotropicMineral` is set to the same volume and temperature. The unrotated cell tensor of the `AnisotropicMineral` ($\mathbf{M}^{\mathrm{h}}(V,T)$) and its scalar pressure ($P^{\mathrm{h}}(V,T)$) are computed. The deformation of the `HyperelasticMineral` away from this hydrostatic cell can then be described by the **isochoric deformation gradient** $\mathbf{F}_{\mathrm{iso}}$ and the **right Hencky strain** $\mathbf{H}$:

$$\mathbf{F}_{\mathrm{iso}} = \mathbf{M}(\mathbf{M}^{\mathrm{h}})^{-1} = \mathbf{R}\mathbf{U}, \qquad \mathbf{H} = \ln \mathbf{U} = \tfrac{1}{2}\ln(\mathbf{F}_{\mathrm{iso}}^{\mathsf{T}}\mathbf{F}_{\mathrm{iso}}).$$

Here $\mathbf{R}$ is a proper rotation, $\mathbf{U}$ is the symmetric positive-definite right stretch, and the logarithm is a matrix logarithm. Because both cells have the same volume,

$$\det \mathbf{F}_{\mathrm{iso}} = 1, \qquad \operatorname{tr} \mathbf{H} = 0, \qquad \mathbf{M} = \mathbf{R}\exp(\mathbf{H})\mathbf{M}^{\mathrm{h}}.$$

The **right, isochoric Hencky strain** can be returned by the `hencky_strain` property. The total deformation gradient $\mathbf{F} = \mathbf{M}\mathbf{M}_0^{-1}$ is returned by the `deformation_gradient_tensor` property. Finally, the rotation matrix $\mathbf{R}$ is returned by the `rotation_matrix` property if the `set_output_frame("current")` method has been called.

4.4.2 Helmholtz equation of state

The constitutive equation for the non-hydrostatic part of the Helmholtz free energy is chosen to be quadratic in the isochoric Hencky strain at fixed volume and temperature:

$$\mathcal{F}(\mathbf{M},T) = \mathcal{F}^{\mathrm{h}}(V,T) + \mathcal{F}^{\mathrm{el}}(\mathbf{M},T), \qquad \mathcal{F}^{\mathrm{el}} = \frac{V}{2} H_{ij}\mathbb{B}^{T,\mathrm{h}}_{ijkl}(V,T)H_{kl}. \tag{4.1}$$

$\mathbb{B}^{T,\mathrm{h}}$ is the hydrostatic mineral's **spatial isothermal stress–strain stiffness**, expressed in the same unrotated axes as $\mathbf{H}$. The hydrostatic cell and stiffness come from the anisotropic equation of state [Myh22]. At $\mathbf{H} = \mathbf{0}$, all thermo-physical properties reduce to the hydrostatic values, with $\boldsymbol{\sigma} = -P^{\mathrm{h}}\mathbf{I}$. The molar volume V converts the energy density into an energy per mole.

4.4.2.1 Why the hydrostatic spatial stiffness appears

The appearance of the spatial stiffness $\mathbb{B}^{T,\mathrm{h}}$ in the hyperelastic model may appear surprising; normally, one would expand the Helmholtz energy in terms of Green–Lagrange strain $\mathbf{E}$ ($\mathbf{E} = (\mathbf{F}^{\mathsf{T}}\mathbf{F} - \mathbf{I})/2$), and therefore

use the material stiffness $\mathbb{C}^T$. This form actually arises directly from an expansion in terms of $\mathbf{E}$, as we now show. The following derivation can be deduced from the description in [Wal67].

Choose one hydrostatic state as a reference, with pressure $P^{T,\mathrm{h}}$ and molar volume V_0. Write $\mathbb{C}^{T,\mathrm{h}}$ for the Hessian of $\Psi = \mathcal{F}/V_0$ with respect to Green–Lagrange strain $\mathbf{E}$. At fixed temperature, the linear coefficient is the reference second Piola–Kirchhoff stress, equal to $-P^{T,\mathrm{h}}\mathbf{I}$. Therefore,

$$\Psi = \Psi^{T,\mathrm{h}} - P^{T,\mathrm{h}} \operatorname{tr}\mathbf{E} + \tfrac{1}{2}\mathbf{E} : \mathbb{C}^{T,\mathrm{h}} : \mathbf{E} + O(\|\mathbf{E}\|^3).$$

The Green–Lagrange strain can itself be expressed in terms of the isochoric Hencky strain $\mathbf{H}$:

$$\mathbf{E} = \tfrac{1}{2}(\exp(2\mathbf{H}) - \mathbf{I}) = \mathbf{H} + \mathbf{H}^2 + O(\|\mathbf{H}\|^3).$$

Even when $\mathbf{H}$ is traceless, $\mathbf{E}$ generally is not. Substituting this expansion and retaining terms through second order gives

$$\Psi = \Psi^{T,\mathrm{h}} - P^{T,\mathrm{h}} \operatorname{tr}\mathbf{H} - P^{T,\mathrm{h}} \operatorname{tr}(\mathbf{H}^2) + \tfrac{1}{2}\mathbf{H} : \mathbb{C}^{T,\mathrm{h}} : \mathbf{H} + O(\|\mathbf{H}\|^3).$$

The hydrostatic stress correction is [Wal67] (Eq. 2.36):

$$\mathbb{B}^{T,\mathrm{h}}_{ijkl} = \mathbb{C}^{T,\mathrm{h}}_{ijkl} - P^{T,\mathrm{h}}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk} - \delta_{ij}\delta_{kl}).$$

Substitution, using symmetric $\mathbf{H}$, gives

$$\Psi = \Psi^{T,\mathrm{h}} - P^{T,\mathrm{h}} \left[\operatorname{tr}\mathbf{H} + \tfrac{1}{2}(\operatorname{tr}\mathbf{H})^2\right] + \tfrac{1}{2}\mathbf{H} : \mathbb{B}^{T,\mathrm{h}} : \mathbf{H} + O(\|\mathbf{H}\|^3).$$

For an isochoric deformation the trace terms vanish. Multiplying by V then yields the quadratic elastic term in (4.1). BurnMan adopts this quadratic dependence at each hydrostatic V, T. Neglecting higher powers of the isochoric Hencky strain is a **constitutive assumption**. It does not imply that the Green–Lagrange energy Hessian remains constant as the isochoric strain increases, either in the model, or in the real system.

4.4.3 Stress, entropy and numerical derivatives

For a symmetric increment $\boldsymbol{\varepsilon}$, define

$$\mathbf{M}(\boldsymbol{\varepsilon}) = \exp(\boldsymbol{\varepsilon})\mathbf{M}, \qquad \widehat{\mathcal{F}}(\boldsymbol{\varepsilon}, T) = \mathcal{F}(\mathbf{M}(\boldsymbol{\varepsilon}), T).$$

Derivatives are evaluated for the deformed cell at $\boldsymbol{\varepsilon} = \mathbf{0}$. They are not derivatives with respect to $\mathbf{H}$. Mechanical work and the entropy definition give

$$d\mathcal{F} = -S\,dT + V\boldsymbol{\sigma} : d\boldsymbol{\varepsilon}, \qquad \sigma_{ij} = \frac{1}{V}\widehat{\mathcal{F}}_{,\varepsilon_{ij}}, \qquad S = -\left(\frac{\partial \mathcal{F}}{\partial T}\right)_{\mathbf{M}}.$$

Holding the physical cell fixed is essential. A temperature change at fixed cell generally changes $\mathbf{M}^{\mathrm{h}}$, and hence $\mathbf{H}$. A strain perturbation can change V, $\mathbf{M}^{\mathrm{h}}$ and $\mathbb{B}^{T,\mathrm{h}}$. For every perturbed state BurnMan recomputes the hydrostatic reference and the polar decomposition; it does not add strain increments directly to $\mathbf{H}$, since strains generally do not commute.

For example, for a symmetric direction $\mathbf{D}$, let $\mathbf{M}_{\pm} = \exp(\pm h\mathbf{D})\mathbf{M}$. The central differences used for the elastic contribution have the form

$$\boldsymbol{\sigma} : \mathbf{D} \simeq -P^{\mathrm{h}} \operatorname{tr}\mathbf{D} + \frac{\mathcal{F}^{\mathrm{el}}(\mathbf{M}_+, T) - \mathcal{F}^{\mathrm{el}}(\mathbf{M}_-, T)}{2hV},$$
$$S \simeq S^{\mathrm{h}} - \frac{\mathcal{F}^{\mathrm{el}}(\mathbf{M}, T + \Delta T) - \mathcal{F}^{\mathrm{el}}(\mathbf{M}, T - \Delta T)}{2\Delta T}.$$

The volume in the stress denominator belongs to the unperturbed state; the volumes inside $\mathcal{F}^{\mathrm{el}}$ belong to their respective perturbed states. The hydrostatic contributions are obtained from the underlying equation of state rather than from differences of the full Helmholtz energy.

The reported `pressure` is defined as the negative one-third of the trace of the Cauchy stress, $P = -\operatorname{tr}\boldsymbol{\sigma}/3$. It need not equal $P^{\mathrm{h}}(V,T)$ away from hydrostatic conditions.

4.4.4 Elastic stiffnesses

BurnMan's `isothermal_stiffness_tensor` is Wallace's $\mathbb{B}^T$, the local response of the Cauchy stress to symmetric spatial strain at fixed temperature and zero incremental rotation [Wal67] (Eqs. 2.35–2.39):

$$d\sigma_{ij} = \mathbb{B}^T_{ijkl}\, d\varepsilon_{kl}, \qquad \mathbb{B}^T_{ijkl} = \frac{1}{V}\widehat{\mathcal{F}}_{,\varepsilon_{ij}\varepsilon_{kl}} - \sigma_{ij}\delta_{kl}.$$

The first term on the RHS of the second equation is the Hessian of the **local exponential strain parameterization** defined in the previous section. The last term accounts for the changing volume in the definition of Cauchy stress. It follows that

$$\mathbb{B}^T_{ijkl} - \mathbb{B}^T_{klij} = \sigma_{kl}\delta_{ij} - \sigma_{ij}\delta_{kl}.$$

Thus $\mathbb{B}^T$ has both minor symmetries but generally lacks major symmetry under non-hydrostatic stress. Symmetrizing its Voigt matrix would change the physical stress response, and so this is not done in BurnMan.

4.4.4.1 Relation to material elasticity

For a fixed original reference, let $\mathbf{F}$ be the total deformation gradient, $J = \det\mathbf{F}$, and $\Psi = \mathcal{F}/V_0$. The work-conjugate stresses are

$$\mathbf{P} = \frac{\partial\Psi}{\partial\mathbf{F}} = J\boldsymbol{\sigma}\mathbf{F}^{-\mathsf{T}}, \qquad \mathbf{S}_{\mathrm{PK2}} = \frac{\partial\Psi}{\partial\mathbf{E}} = J\mathbf{F}^{-1}\boldsymbol{\sigma}\mathbf{F}^{-\mathsf{T}}.$$

The material Hessian $\mathbb{C}^T_{IJKL} = \partial^2\Psi/\partial E_{IJ}\partial E_{KL}$ must first be pushed forward into the current configuration:

$$\mathbb{C}^{E,T}_{ijkl} = \frac{1}{J}F_{iI}F_{jJ}F_{kK}F_{lL}\mathbb{C}^T_{IJKL}.$$

Wallace's stress correction then gives

$$\mathbb{B}^T_{ijkl} = \mathbb{C}^{E,T}_{ijkl} + \tfrac{1}{2}\left(\sigma_{ik}\delta_{jl} + \sigma_{il}\delta_{jk} + \sigma_{jk}\delta_{il} + \sigma_{jl}\delta_{ik} - 2\sigma_{ij}\delta_{kl}\right). \tag{4.2}$$

When the current configuration is chosen as the reference, $\mathbf{F} = \mathbf{I}$ and $\mathbb{C}^{E,T} = \mathbb{C}^T$ at that state. This choice does not require the total prior deformation to be small. For a general incremental displacement gradient $\mathbf{L} = \mathbf{D} + \mathbf{W}$, with symmetric $\mathbf{D}$ and antisymmetric $\mathbf{W}$, the stress increment also includes rotation:

$$d\boldsymbol{\sigma} = \mathbb{B}^T : \mathbf{D} + \mathbf{W}\boldsymbol{\sigma} - \boldsymbol{\sigma}\mathbf{W} + \boldsymbol{\pi}\, dT.$$

4.4.4.2 Seismic wave propagation

For small, locally plane, adiabatic waves about a homogeneous prestressed state, Wallace's equation-of-motion coefficients are [Wal67] (Eqs. 2.24 and 2.28)

$$\mathbb{A}^S_{ijkl} = \mathbb{C}^{E,S}_{ijkl} + \delta_{ik}\sigma_{jl}, \qquad T_{ik} = \mathbb{A}^S_{ijkl}n_jn_l, \qquad T_{ik}u_k = \rho v^2 u_i.$$

Wallace denotes $\mathbb{A}^S$ by $\mathbb{S}_{ijkl}$ (the symbol S is overused in thermodynamics; this tensor should not be confused with compliance tensors, entropy, or the second Piola–Kirchhoff stress). In this expression, $\mathbf{n}$ is a unit propagation direction in the current frame, ρ is the current mass density, and $\mathbf{u}$ is the polarization. $\mathbb{C}^{E,S}$ is the pushed-forward Green-strain Hessian of internal energy at fixed entropy.

Eliminating $\mathbb{C}^{E,S}$ with (4.2), using its isentropic counterpart, gives the form used by the `christoffel_tensor` method:

$$T^B_{ik} = \mathbb{B}^S_{ijkl} n_j n_l, \qquad T_{ik} = \tfrac{1}{2}(T^B_{ik} + T^B_{ki}) + \tfrac{1}{2}\left[(\mathbf{n}\cdot\boldsymbol{\sigma}\mathbf{n})\delta_{ik} - \sigma_{ik}\right].$$

The Christoffel tensor $\mathbf{T}$ can then be used to compute wave speeds and polarizations with the `wave_velocities` method:

$$\mathbf{T}\mathbf{u}^{(a)} = \lambda_a \mathbf{u}^{(a)}, \qquad v_a = \sqrt{\frac{\lambda_a}{\rho}}, \qquad \|\mathbf{u}^{(a)}\| = 1, \qquad a = 1, 2, 3.$$

Each eigenvalue λ_a gives a wave speed v_a, and its normalized eigenvector $\mathbf{u}^{(a)}$ gives the polarization.

Under hydrostatic stress the additional term vanishes and $\mathbf{T}^B$ is symmetric, and so either $\mathbf{T}^B$ or $\mathbf{T}$ can be used. Under non-hydrostatic stress, $\mathbf{T}$ must be used, as done in the `wave_velocities` method.

4.4.5 Maxwell relations and thermal response

The fixed-cell thermal stress and molar isometric heat capacity are

$$\pi_{ij} = \left(\frac{\partial \sigma_{ij}}{\partial T}\right)_{\mathbf{M}} = \frac{1}{V}\widehat{\mathcal{F}}_{,\varepsilon_{ij}T} = -\frac{1}{V}\left(\frac{\partial S}{\partial \varepsilon_{ij}}\right)_T, \qquad C_\varepsilon = T\left(\frac{\partial S}{\partial T}\right)_{\mathbf{M}}.$$

The Maxwell relation leads to the entropy–strain derivative:

$$dS = \frac{C_\varepsilon}{T}\,dT - V\boldsymbol{\pi} : d\boldsymbol{\varepsilon}.$$

At constant entropy, substituting the resulting temperature increment into the stress increment gives the isentropic stiffness,

$$\mathbb{B}^S_{ijkl} = \mathbb{B}^T_{ijkl} + \frac{VT}{C_\varepsilon}\pi_{ij}\pi_{kl}.$$

For zero incremental rotation, the thermal expansivity tensor at fixed Cauchy stress and the corresponding molar heat capacity are

$$\mathbb{S}^T = (\mathbb{B}^T)^{-1}, \qquad \boldsymbol{\alpha} = -\mathbb{S}^T : \boldsymbol{\pi}, \qquad C_\sigma = C_\varepsilon - VT\boldsymbol{\pi} : \boldsymbol{\alpha}.$$

Here the inverse acts on symmetric tensors. Because $\mathbb{S}^T$ need not have major symmetry, define separately $\widetilde{\alpha}_{kl} = -\pi_{ij}\mathbb{S}^T_{ijkl}$. Then

$$\mathbb{S}^S_{ijkl} = \mathbb{S}^T_{ijkl} - \frac{VT}{C_\sigma}\alpha_{ij}\widetilde{\alpha}_{kl}.$$

Replacing $\widetilde{\boldsymbol{\alpha}}$ by $\boldsymbol{\alpha}$ is valid only when the two contractions agree, as they do for a major-symmetric compliance. The volumetric expansivity is $\alpha_V = \operatorname{tr}\boldsymbol{\alpha}$; the local Reuss compressibility is $\beta^T_R = \delta_{ij}\mathbb{S}^T_{ijkl}\delta_{kl}$ and $K^T_R = 1/\beta^T_R$.

The relevant properties are `thermal_stress`, `thermal_expansivity_tensor`, `molar_isometric_heat_capacity`, `molar_isostress_heat_capacity` and the isothermal/isentropic stiffness and compliance tensors. Isometric means fixed cell shape **and** volume. It is a stronger constraint than fixing volume while allowing the hydrostatic reference cell to change shape.

4.4.6 Rotations and output frames

Rigid rotation changes tensor components but leaves the energy, volume, entropy and wave speeds for correspondingly rotated directions unchanged. For a rotation $\mathbf{Q}$,

$$\mathbf{M}' = \mathbf{Q}\mathbf{M}, \qquad \boldsymbol{\sigma}' = \mathbf{Q}\boldsymbol{\sigma}\mathbf{Q}^{\mathsf{T}}, \qquad \mathbb{B}'_{ijkl} = Q_{ia}Q_{jb}Q_{kc}Q_{ld}\mathbb{B}_{abcd}.$$

`set_output_frame("current")` reports spatial tensors in the frame of the supplied cell vectors; this is the default. `set_output_frame("crystal")` reports them in the crystallographic frame defined by the cell and the underlying mineral's frame convention. This changes the representation, not the physical state. Input to `set_state_with_molar_cell_vectors` is always in the current frame, regardless of the selected output frame. The right `hencky_strain` stays in the hydrostatic reference axes and is not a spatial tensor to be rotated with the output frame.

4.4.7 Using the model

Given an existing `AnisotropicMineral` named `hydrostatic_mineral`, the following applies a volume-preserving simple shear at 1000 K:

```
import numpy as np
from burnman import HyperelasticMineral

hydrostatic_mineral.set_state(5.0e9, 1000.0)
M_h = hydrostatic_mineral.unrotated_cell_vectors.T.copy()
mineral = HyperelasticMineral(hydrostatic_mineral)

F = np.eye(3)
F[0, 1] = 0.05
mineral.set_state_with_molar_cell_vectors((F @ M_h).T, 1000.0)

print("Molar volume (m^3/mol):", mineral.V)
print("Mean pressure (GPa):", mineral.pressure / 1.0e9)
print("Cauchy stress (GPa):\n", mineral.cauchy_stress / 1.0e9)
velocities, polarizations = mineral.wave_velocities([1.0, 0.0, 0.0])
print("Wave speeds (km/s):", velocities / 1.0e3)
```

The supplied hydrostatic object is retained and its state is updated during hyperelastic calculations. Use separate hydrostatic instances for independent hyperelastic models. The repository example `examples/example_hyperelastic_mineral.py` constructs a complete model, solves shear at fixed normal traction, and plots stresses, relative volume change and cell shape.

`burnman.tools.eos.check_hyperelastic_eos_consistency` checks energy derivatives, Maxwell relations, thermal conversions and rotation behaviour. Numerical derivatives have finite step-size and cancellation errors; exact thermodynamic identities should therefore be tested with numerical tolerances.

4.4.8 Class reference

class burnman.classes.hyperelastic.**HyperelasticMineral**(*hydrostatic_mineral*)

Bases: *AnisotropicMineral*

This class represents a hyperelastic mineral, where the word "hyperelastic" indicates that the mineral can undergo large elastic deformations to non-hydrostatic states.

The hydrostatic state of the mineral is represented by an instance of the AnisotropicMineral class. The non-hydrostatic elastic energy at fixed volume and temperature is then calculated as 1/2 * V * H : C^hyd : H, where V is the volume of the mineral, H is the right Hencky strain tensor describing the isochoric deformation of the mineral away from the hydrostatic state, and C^hyd is the *spatial* isothermal stiffness tensor of the mineral at the hydrostatic state. All other thermodynamic properties are calculated as thermodynamic derivatives of the total Helmholtz free energy.

Despite using the *spatial* stiffness tensor, it can be shown that the elastic energy term corresponds to treating the isochoric components of the *material* isothermal stiffness tensor as constant. The non-isochoric components of the material stiffness tensor are not constant, and neither is the spatial stiffness tensor. Indeed, the spatial stiffness tensor does not even retain the major symmetry ijkl = klij under non-hydrostatic conditions.

Setting the state of a HyperelasticMineral instance requires specifying the cell vectors and temperature via the set_state_with_molar_cell_vectors method.

Properties of the mineral can be reported in either the current configuration defined by the cell vectors, or in the crystallographic frame defined by the cell vectors and frame convention of the underlying AnisotropicMineral. By default, properties are reported in the current configuration, but the set_output_frame method can be used to toggle between "current" and "crystal" output frames.

The stiffness tensors output by this class are derivatives of the Cauchy stress with respect to the strain. Under non-hydrostatic conditions, these are not the tensors directly relevant for seismic wave propagation. The christoffel_tensor method of the AnisotropicMineral class applies the required correction, which includes terms involving the Cauchy stress (Wallace, 1967; doi:10.1103/PhysRev.162.776).

CHAPTER

FIVE

RELAXED MATERIAL CLASSES

5.1 Overview

Some materials have isochemical degrees of freedom, meaning that some macroscopic characteristic of the material can change without changing its overall composition. For example, a solid solution may be able to change how Fe and Mg are distributed amongst its constituent sites, or Fe may be able to change its spin state, or the material itself may be able to change by distortion of the crystal lattice. In a multiphase material, isochemical degrees of freedom may arise from changes in the proportions or compositions of the constituent phases (e.g., Fe-Mg exchange). If these isochemical variables can change on timescales shorter than the timescales of changes in pressure and temperature, then the material is said to be relaxed with respect to these isochemical degrees of freedom.

BurnMan provides classes to model relaxed materials, including *burnman.RelaxedSolution*, *burnman.RelaxedAnisotropicSolution*, and *burnman.RelaxedComposite*. These classes extend their unrelaxed counterparts by adding the capability to minimize the Gibbs energy of the system by varying one or more isochemical variables at fixed pressure and temperature. These modified classes are not simply a combination of the unrelaxed classes with a Gibbs minimiser; they also modify the way thermodynamic properties are calculated to account for the relaxation of the isochemical variables.

The properties of particular interest in relaxed materials are those that depend on the second derivatives of the Gibbs energy, such as the bulk modulus, thermal expansivity, and heat capacity. In unrelaxed materials, these properties are calculated directly from the second derivatives of the Gibbs energy with respect to pressure and temperature, keeping all molar fractions constant. By contrast, in relaxed materials, the second derivatives are affected by simultaneous changes in the isochemical variables. To account for this, BurnMan uses variational calculus to derive expressions for the relaxed second derivatives of the Gibbs energy, which are then used to compute the thermodynamic properties of the relaxed material.

5.2 Mathematical Derivation

Consider a material whose state is determined by the variables pressure P, temperature T and amounts of independent components C. Additionally, suppose that C can be decomposed into a set of variables X, which are externally controlled either by conservation of composition or because they evolve slowly compared to changes in P and T, and a set of isochemical variables ξ, which can change rapidly at fixed P, T, and X. These variables ξ are internal variables, and might represent (for example) a change in iron spin state, state of Fe-Mg order, or extent of Fe-Mg exchange between phases.

The driving force for changes in the isochemical variables is the minimization of the Gibbs free energy $\mathcal{G}(P, T, X, \xi)$. Let us define the partial derivative of the Gibbs energy with respect to the isochemical vari-

ables as $F(P,T,X,\xi) = \partial\mathcal{G}(P,T,X,\xi)/\partial\xi$, and for simplicity, let us also group the pressure, temperature, and composition variables into a single vector $Z = \{P,T,X\}$, such that

$$F(Z,\xi) = \left(\partial\mathcal{G}(Z,\xi)/\partial\xi\right)_Z .$$

Now, at equilibrium, the Gibbs energy is minimized with respect to the isochemical variables, such that

$$F(Z,\xi) = 0.$$

Let us now define another function $\phi(Z) = \xi$, which represents the values of the isochemical variables that minimize the Gibbs energy at fixed Z.

This yields the modified function

$$f(Z) = F(Z,\phi(Z)) = \frac{\partial\mathcal{G}}{\partial\xi}(Z,\phi(Z)),$$

where the lowercase f indicates that this function depends only on Z. Taking the total derivative of this expression with respect to Z, we have:

$$\frac{df}{dZ} = \frac{\partial F}{\partial Z} + \frac{\partial F}{\partial\xi}\frac{\partial\phi}{\partial Z}$$

Now, since $f(Z) = 0$ for all Z, we also have $df/dZ = 0$, and therefore:

$$\frac{\partial F}{\partial\xi}\frac{\partial\phi}{\partial Z} = -\frac{\partial F}{\partial Z}$$

Note that the matrix $\partial F/\partial\xi$ is the Hessian matrix of second derivatives of the Gibbs energy with respect to the isochemical variables:

$$\frac{\partial F}{\partial\xi} = \frac{\partial^2\mathcal{G}}{\partial\xi^2}$$

It is therefore symmetric. Let us define the pseudo-inverse of this matrix as R, so that $R\partial F/\partial\xi = I$, where I is the identity matrix. Then:

$$\frac{\partial\phi}{\partial Z} = -R\frac{\partial F}{\partial Z}$$

Finally, define a function $g(Z) = \mathcal{G}(Z,\phi(Z))$, which gives the Gibbs energy of the system at equilibrium. Note that g has the same value as $\mathcal{G}$, but, like f, depends only on Z, since the value of ξ is determined by minimization. Using the chain rule, we can write the total derivative of this function with respect to Z as:

$$\frac{dg}{dZ} = \frac{\partial\mathcal{G}}{\partial Z} + \frac{\partial\mathcal{G}}{\partial\xi}\frac{\partial\phi}{\partial Z}$$

Since $F = \partial\mathcal{G}/\partial\xi = 0$ at equilibrium, the second term on the right-hand side vanishes, and we have:

$$\frac{dg}{dZ} = \frac{\partial\mathcal{G}(Z,\phi(Z))}{\partial Z}$$

Taking the second derivative of g with respect to Z:

$$\frac{d^2g}{dZ^2} = \frac{\partial^2\mathcal{G}}{\partial Z^2} + \frac{\partial^2\mathcal{G}}{\partial Z\partial\xi}\frac{\partial\phi}{\partial Z}$$

Now, substituting in our expression for $\partial\phi/\partial Z$:

$$\frac{d^2g}{dZ^2} = \frac{\partial^2\mathcal{G}}{\partial Z^2} - \frac{\partial^2\mathcal{G}}{\partial Z\partial\xi}R\frac{\partial F}{\partial Z}$$

or, equivalently:

$$\frac{d^2g}{dZ^2} = \frac{\partial^2\mathcal{G}}{\partial Z^2} - \frac{\partial^2\mathcal{G}}{\partial Z\partial\xi}R\frac{\partial^2\mathcal{G}}{\partial\xi\partial Z}$$

This expression gives the relaxed second derivatives of the Gibbs energy with respect to pressure, temperature, and composition. These second derivatives can then be used to calculate the relaxed thermodynamic properties of the system, such as the bulk modulus, thermal expansivity, and heat capacity:

$$K_{P,relaxed} = -V\left(\frac{\partial^2 g}{\partial P^2}\right)^{-1}, \quad \alpha_{relaxed} = \frac{1}{V}\frac{\partial^2 g}{\partial P\partial T}, \quad C_{P,relaxed} = -T\frac{\partial^2 g}{\partial T^2}$$

5.3 Relaxed Material Classes

5.3.1 Relaxed Solution Class

class burnman.**RelaxedSolution**(*solution*, *relaxation_vectors*, *unrelaxed_vectors*)

Bases: *Solution*

A class implementing a relaxed solution model. This class is derived from Solution, and inherits most of the methods from that class.

Instantiation of a RelaxedSolution involves passing an Solution, plus a set of vectors that represent rapid deformation modes. For example, a solution of MgO, FeHSO and FeLSO (high and low spin wuestite) can rapidly change proportion of high spin and low spin iron, and so a single vector should be passed: np.array([[0., -1., 1.]]) or some multiple thereof.

States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.RelaxedSolution`.

5.3.2 Relaxed Anisotropic Solution Class

class burnman.**RelaxedAnisotropicSolution**(*anisotropic_solution*, *relaxation_vectors*, *unrelaxed_vectors*)

Bases: *AnisotropicSolution*

A class implementing the relaxed anisotropic solution model described in [Myh25b]. This class is derived from AnisotropicSolution, and inherits most of the methods from that class.

Instantiation of a RelaxedAnisotropicSolution involves passing an AnisotropicSolution, plus a set of vectors that represent rapid deformation modes. For example, a solution of MgO, FeHSO and FeLSO (high and low spin wuestite) can rapidly change proportion of high spin and low spin iron, and so a single vector should be passed: np.array([[0., -1., 1.]]) or some multiple thereof.

States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.RelaxedAnisotropicSolution`.

5.3.3 Relaxed Composite Class

class burnman.**RelaxedComposite**(*composite*, *relaxation_vectors*)

Bases: *Composite*

A class implementing a relaxed Composite mineral assemblage. Unlike a standard Composite, the set_state() method of this class uses the *burnman.equilibrate* function to find the equilibrium phase fractions and compositions at given pressure and temperature. Once equilibrated, the thermodynamic properties of the relaxed composite can be queried in the same way as for a standard Composite.

One additional feature of this class is that properties related to the second derivatives of the Gibbs energy (e.g. compressibility, thermal expansivity, heat capacity) can be calculated taking into account relaxation of one or more user-defined isochemical reaction vectors via the relaxation_vectors argument to the constructor. If complete relaxation is desired, the user can pass relaxation_vectors=composite.reaction_basis. More limited relaxation can be specified by passing a linear subset of the reaction basis vectors.

For example, on short timescales,a composite of olivine and wadsleyite might be able to exchange Mg and Fe, even if no grain growth or reduction can occur. In this case, a single vector representing the exchange of Mg and Fe between the two minerals can be used to relax the composition at given P and T: relaxation_vectors=[[1, -1, -1, 1]], where the first two entries correspond to olivine (fo, fa) and the second two to wadsleyite (mwd, fwd). This is a subset of the full reaction basis, which would be: [[1, 0, -1, 0], [0, 1, 0, -1]].

Example:

```
from burnman import RelaxedComposite

# Create a standard Composite
# This is also where you would set the composition of
# the individual phases if desired.
unrelaxed_composite = Composite(...) # see Composite documentation

# Create a RelaxedComposite that reacts quickly relative to
# changes in pressure and temperature
composite = RelaxedComposite(
    unrelaxed_composite,
    relaxation_vectors=unrelaxed_composite.reaction_basis
)

# Set the state of the relaxed composite at 10 GPa and 1500 K
# This will equilibrate the composite at its current bulk
# composition.
composite.set_state(1.0e10, 1500.0)

# Now do the same but specifying a different bulk composition
bulk_composition = {'Mg': 0.8, 'Fe': 0.2, 'Si': 1.0, 'O': 4.0}
composite.set_state(1.0e10, 1500.0, bulk_composition=bulk_composition)
```

(continues on next page)

(continued from previous page)

```
# Query thermodynamic properties as usual
K_T = composite.isothermal_bulk_modulus_reuss
```

Parameters

- **composite** (*burnman.Composite*) – The unrelaxed Composite material.
- **relaxation_vectors** (*list of lists or 2D array*) – A list of isochemical relaxation vectors. Each vector should be a list or array with length equal to the number of endmembers in the composite. The number of vectors determines the number of degrees of freedom available for relaxation.

CHAPTER

SIX

AVERAGING SCHEMES

6.1 Overview

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:

$$V = \sum_i n_i V_i.$$

The density of the multiphase assemblage is then

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

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 effective 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 effective elastic moduli K_{eff} and G_{eff}: 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

$$X_V = \sum_i \nu_i X_i,$$

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

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

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

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

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.

6.2 Implemented Averaging Schemes

6.2.1 VoigtReussHill

class burnman.classes.averaging_schemes.**VoigtReussHill**

Bases: *AveragingScheme*

Class for computing the Voigt-Reuss-Hill average for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

6.2.2 Voigt

class burnman.classes.averaging_schemes.**Voigt**

Bases: *AveragingScheme*

Class for computing the Voigt (iso-strain) bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

6.2.3 Reuss

class burnman.classes.averaging_schemes.**Reuss**

Bases: *AveragingScheme*

Class for computing the Reuss (iso-stress) bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

6.2.4 HashinShtrikmanUpper

class burnman.classes.averaging_schemes.**HashinShtrikmanUpper**

Bases: *AveragingScheme*

Class for computing the upper Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

6.2.5 HashinShtrikmanLower

class burnman.classes.averaging_schemes.**HashinShtrikmanLower**

Bases: *AveragingScheme*

Class for computing the lower Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements Formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

6.2.6 HashinShtrikmanAverage

class burnman.classes.averaging_schemes.**HashinShtrikmanAverage**

Bases: *AveragingScheme*

Class for computing arithmetic mean of the Hashin-Shtrikman bounds on elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

6.3 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:

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

6.4 Attenuation

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] using the function *burnman.tools.seismic.attenuation_correction()*:

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

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].

CHAPTER

SEVEN

MINERAL DATASETS

BurnMan includes a large collection of mineral datasets in the the `burnman.minerals` module that provide thermodynamic and thermoelastic properties for minerals commonly found in the Earth and planetary interiors.

See *Mineral databases* for a list of built-in minerals and solutions available in BurnMan.

CHAPTER

EIGHT

CALIBRANTS

Sometimes, it is unnecessary or inconvenient to define a full material model to represent a substance. One of these cases is when dealing with experimental calibrants, where we are only interested in finding the pressure and/or temperature conditions based on measured unit cell volumes. BurnMan includes a Calibrant class for this purpose. A Calibrant is defined by a volumetric equation of state (EOS) and accompanying parameters. It may or may not be temperature dependent.

See *Calibrant databases* for a list of built-in calibrants available in BurnMan.

CHAPTER

NINE

SEISMIC MODELS

9.1 Overview

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.

See *Seismic Models* for a list of built-in seismic models available in BurnMan.

9.2 Comparison to Seismic profiles

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

TEN

GEOTHERMS

Unlike the pressure, the temperature of the lower mantle is relatively unconstrained. BurnMan provides lightweight `burnman.classes.geotherm.Geotherm` and `burnman.classes.geotherm.GeothermFromPressures` classes that store a geotherm defined by arrays of depths and temperatures or pressures and temperatures respectively. Both classes have a *temperatures()* method that takes an array or list of depths as input and outputs the corresponding temperatures along the geotherm. In addition, the `burnman.classes.geotherm.GeothermFromPressures` class has a *temperatures_from_pressures()* method that takes an array or list of pressures as input. A number of geotherms from the literature are provided. See *Geotherms* for a list.

Also implemented in BurnMan is an `burnman.classes.geotherm.AdiabaticGeotherm` which allows for the computation of adiabatic temperature profiles given a pressure to depth conversion, a material, and an anchor temperature and pressure. Again, see *Geotherms* for more details.

CHAPTER

ELEVEN

COMPOSITION CLASS

class burnman.classes.composition.**Composition**(*composition_dictionary*, *unit_type='mass'*, *normalize=False*)

Bases: `object`

The Composition class can be used to store, modify and renormalize compositions, and also convert between mass, molar and atomic amounts. Weight is provided as an alias for mass, as we assume that only Earthlings will use this software.

Composition can parse dictionaries of components, rather than only elements. Components are given as strings (e.g. 'MgSiO3', 'Fe2O3', etc). The Composition class breaks these components down into their elemental constituents internally.

Example:

```
from burnman.composition import Composition

comp_dict = {'MgSiO3': 0.5, 'FeSiO3': 0.5}
comp = Composition(comp_dict, unit_type='molar', normalize=True)
print(comp.atomic_composition)

# Renormalize to 1 kg of material
comp.renormalize(unit_type='mass', normalization_component='total',␣
↪normalization_amount=1.0)
print(comp.mass_composition)

# Add 0.1 moles of Al2O3 to the composition
comp.add_components({'Al2O3': 0.1}, unit_type='molar')
print(comp.molar_composition)

# Change the component set to only oxides
comp.change_component_set(['MgO', 'SiO2', 'FeO', 'Al2O3'])
print(comp.molar_composition)

# Remove components with less than 1.e-10 moles
comp.remove_null_components(tol=1.e-10)
print(comp.molar_composition)
```

(continues on next page)

(continued from previous page)

```
# Pretty-print the molar composition, normalized to 100 moles total
comp.print(unit_type='molar', significant_figures=3, normalization_
↪component='total', normalization_amount=100.0)
```

Parameters

- **composition_dictionary** (*dictionary*) – Dictionary of components (given as a string) and their amounts.
- **unit_type** (*str*) – 'mass', 'weight' or 'molar' (optional, 'mass' as default) Specify whether the input composition is given as mass or molar amounts.
- **normalize** (*bool*) – If False, absolute numbers of kilograms/moles of component are stored, otherwise the component amounts of returned compositions will sum to one (until Composition.renormalize() is used).

CHAPTER

TWELVE

LAYERS AND PLANETS

Please refer to the API documentation on *Layers and Planets*.

CHAPTER

THIRTEEN

MATERIAL POLYTOPES

Often in mineral physics, material compositions, solution compositions, or solution site-occupancies 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.

Please see *Polytopes* for full documentation.

CHAPTER

FOURTEEN

FITTING MODEL PARAMETERS TO DATA

BurnMan provides several tools for fitting model parameters to experimental or observational data. These include:

- calculation of phase proportions in an assemblage given a bulk composition and phase compositions
- calculation of independent endmember proportions in a solution given its composition
- **fitting equations of state parameters to P-T data including volumes,**
  heat capacities and seismic velocities.
- **fitting solution model parameters to P-T-compositional data, also including volumes,**
  heat capacities and seismic velocities.

In all cases, the optimal solution is found in a weighted least-squares sense, taking into account uncertainties in the data where provided, and propogating uncertainties into the fitted parameters. Data uncertainties can include full covariance matrices in P, T, V, composition space, etc. Prior estimates of parameters can also be provided.

14.1 Fitting Model Compositions to Data

14.1.1 Overview

In experimental petrology and geochemistry, it is often useful to:

- **fit proportions of mineral phases to bulk rock compositions.**
  This can be used, for example, to perform mass balance calculations to check if an experimental run product is consistent with the starting composition.
- **fit molar fractions of solution endmembers to analysed elemental or oxide compositions.**
  This can form a starting point for *Thermobarometry*, or for fitting parameters of thermodynamic datasets.

BurnMan provides functions to perform these types of composition fitting.

14.1.2 Available Fitting Functions

burnman.composition_fitting.**fit_phase_proportions_to_bulk_composition**(*phase_compositions*, *bulk_composition*)

Determine the proportions of phases that best fit a given bulk composition (in a weighted constrained least squares sense).

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

(phase_proportions, covariance_matrix, weighted_residual)

- phase_proportions: Optimized proportions of each phase.
- **covariance_matrix: Covariance matrix of the optimized phase proportions,** computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing the square root** of the weighted sum of squared differences between observed and modeled bulk compositions.

Return type

tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

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

Fit a set of molar fractions to a solution model that best matches observed compositional variables (in a least-squares sense).

The fitting applies appropriate non-negativity constraints (i.e. no species can have a negative occupancy on a site).

Parameters

- **solution** (*burnman.Solution*) – The solution to use in the fitting procedure.
- **fitted_variables** (*list of str*) – A list of the variables used to find the best-fit molar fractions of the solution. These should either be elements such as "Fe", site_species such as "Fef_B" which would correspond to a species labelled Fef on the second site, or user-defined variables which are arithmetic sums of elements and/or site_species defined in "variable_conversions".

- **variable_values** (*numpy.array*) – Numerical values of the fitted variables. These should be given as amounts; they do not need to be normalized.
- **variable_covariances** (*2D numpy.array*) – Covariance matrix of the variables.
- **variable_conversions** (*dict of dict, or None*) – A dictionary converting any user-defined variables into an arithmetic sum of element and site-species amounts. For example, {'Mg_equal': {'Mg_A': 1., 'Mg_B': -1.}}, coupled with Mg_equal = 0 would impose a constraint that the amount of Mg would be equal on the first and second site in the solution.
- **normalize** (*bool, default=True*) – Whether to normalize the molar fractions to sum to one.

Returns

(molar_fractions, covariance_matrix, weighted_residual):

- molar_fractions: Optimized molar fractions of each endmember.
- **covariance_matrix: Covariance matrix of the optimized molar** fractions, computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing** the square root of the weighted sum of squared differences between observed and modeled variables.

Return type

tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

14.2 Fitting Equation of State Parameters to Data

14.2.1 Overview

BurnMan implements completely general nonlinear least squares fitting routines in the function `burnman.optimize.eos_fitting.nonlinear_least_squares_fit()`. This function takes an instance of a *burnman.optimize.nonlinear_fitting.NonLinearModel* as input, along with data to be fit, and returns best-fit parameters along with uncertainties.

For many users, the process of constructing the *NonLinearModel* class may be unfamiliar and tedious. To facilitate fitting equations of state to data, BurnMan provides a number of helper functions that construct *NonLinearModel* instances so that users can easily fit equations of state to data.

14.2.2 Available Fitting Functions

burnman.optimize.eos_fitting.**fit_PTV_data**(*mineral*, *fit_params*, *data*, *data_covariances=[]*, *delta_params=None*, *bounds=None*, *param_tolerance=1e-05*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

A simple alias for the fit_PTp_data for when all the data is volume data

burnman.optimize.eos_fitting.**fit_PTp_data**(*mineral*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a mineral of any type, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the parameters should be optimized.
- **fit_params** (*list of str*) – List of dictionary keys contained in mineral.params corresponding to the variables to be optimized during fitting. Initial guesses are taken from the existing values for the parameters
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional) If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).

- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns
Model with optimized parameters.

Return type
burnman.optimize.eos_fitting.MineralFit

burnman.optimize.eos_fitting.**fit_VTP_data**(*mineral*, *fit_params*, *data*, *data_covariances=[]*, *delta_params=None*, *bounds=None*, *param_tolerance=1e-05*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

A simple alias for the fit_VTp_data for when all the data is pressure data

burnman.optimize.eos_fitting.**fit_VTp_data**(*mineral*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a mineral of any type, a list of fit parameters and a set of V-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the parameters should be optimized.
- **fit_params** (*list of str*) – List of dictionary keys contained in mineral.params corresponding to the variables to be optimized during fitting. Initial guesses are taken from the existing values for the parameters
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'P')
- **data** (*2D numpy.array*) – Observed V-T-property values
- **data_covariances** (*3D numpy.array*) – V-T-property covariances (optional) If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the properties.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).

- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns
Model with optimized parameters.

Return type
burnman.optimize.eos_fitting.MineralFitV

14.3 Fitting Solution Model Parameters to Data

14.3.1 Overview

BurnMan implements completely general nonlinear least squares fitting routines in the function `burnman.optimize.eos_fitting.nonlinear_least_squares_fit()`. This function takes an instance of a *burnman.optimize.nonlinear_fitting.NonLinearModel* as input, along with data to be fit, and returns best-fit parameters along with uncertainties.

For many users, the process of constructing the *NonLinearModel* class may be unfamiliar and tedious. To facilitate fitting solution parameters to data, BurnMan provides a helper function that constructs *NonLinearModel* instances so that users can easily fit solution parameters to data.

14.3.2 Available Fitting Functions

burnman.optimize.eos_fitting.**fit_XPTp_data**(*solution*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a symmetric solution, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **solution** (*burnman.Solution*) – Solution for which the parameters should be optimized.

- **fit_params** (*list of lists*) – Variables to be optimized during fitting. Each list is either of length two or three. The first item of length-2 lists should be a dictionary key contained in one of the endmember mineral.params, and the second item should be the index of the endmember in the solution (indexing starts from 0). The first item of length-3 lists should be one of 'E', 'S' or 'V' (the excess energies, entropies or volumes in each binary). The second two items should be the indices of the pair of endmembers bounding the binary, in ascending order (indexing starts from 0). Initial guesses are taken from the existing values for the parameters.
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional). If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns

Model with optimized parameters.

Return type

burnman.optimize.eos_fitting.SolutionFit

CHAPTER

FIFTEEN

EQUILIBRATION

15.1 Overview

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 `burnman.equilibrate()` function.

15.2 Implemented function

`burnman.`**`equilibrate`**(*composition*, *assemblage*, *equality_constraints*, *free_compositional_vectors=[]*, *tol=None*, *store_iterates=False*, *store_assemblage=True*, *max_iterations=100.0*, *verbose=False*)

A function that finds the thermodynamic equilibrium state of an assemblage subject to given equality constraints by solving a set of nonlinear equations related to the chemical potentials and other state variables of the system.

The user chooses an assemblage (e.g. olivine, garnet and orthopyroxene) and $2 + n_c$ equality constraints, where n_c is the number of bulk compositional degrees of freedom. The equilibrate function attempts to find the remaining unknowns that satisfy those constraints.

There are a number of equality constraints implemented in burnman. These are given as a list of lists. Each constraint should have the form: [<constraint type>, <constraint>], where <constraint type> is one of 'P', 'T', 'S', 'V', 'X', 'phase_fraction', or 'phase_composition'. The format of the <constraint> object depends on the constraint type:

- **P: float or numpy.array of**
  pressures [Pa]
- **T: float or numpy.array of**
  temperatures [K]

- **S: float or numpy.array of**
  entropies [J/K]
- **V: float or numpy.array of**
  volumes [m:math:^3]
- **phase_fraction: tuple with the form (<phase> <fraction(s)>),**
  where <phase> is one of the phase objects in the assemblage and <fraction(s)> is a float or numpy.array corresponding to the desired phase fractions.
- **phase_composition: tuple with the form (<phase> <constraint>),**
  where <phase> is one of the phase objects in the assemblage and <constraint> has the form (site_names, n, d, v), where $(nx)/(dx) = v$, n and d are constant vectors of site coefficients, and v is a float or numpy.array. For example, a constraint of the form ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) would correspond to equal amounts Mg and Fe on the A site (Mg_A / (Mg_A + Fe_A) = 0.5).
- **X: list of a numpy.array and a float or numpy.array,**
  where the equality satisfies the linear equation arr[0] x = arr[1]. The first numpy.array is fixed, and the second is to be looped over by the equilibrate function. This is a generic compositional equality constraint.

Parameters

- **composition** (*dict*) – The bulk composition that the assemblage must satisfy.
- **assemblage** (*burnman.Composite* or *burnman.RelaxedComposite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of list*) – The list of equality constraints. See above for valid formats.
- **free_compositional_vectors** (*list of dict*) – A list of dictionaries containing the compositional freedom of the solution. For example, if the list contains the vector {'Mg': 1., 'Fe': -1}, that implies that the bulk composition is equal to composition + a (n_Mg - n_Fe), where the value of a is to be determined by the solve. Vector given in atomic (molar) units of elements.
- **tol** (*float or numpy.array*) – The tolerance for the nonlinear solver. This can be a single float, which is then applied to all parameters, or a numpy array with the same length as the number of parameters in the solve (2 + number of phases + number of free compositional vectors). The default is None, which sets the tolerances to 1 Pa for pressure, 1e-3 K for temperature, 1e-6 for phase amounts and 1e-6 for free compositional vectors. One of the termination criteria for the nonlinear solver is that the absolute change in each parameter is less than the corresponding tolerance.
- **store_iterates** (*bool*) – Whether to store the parameter values for each iteration in each solution object.
- **store_assemblage** (*bool*) – Whether to store a copy of the assemblage object in each solution object.

- **max_iterations** (*int*) – The maximum number of iterations for the nonlinear solver.
- **verbose** (*bool*) – Whether to print output updating the user on the status of equilibration.

Returns

Solver solution object (or a list, or a 2D list of solution objects) created by *burnman.optimize.nonlinear_solvers.damped_newton_solve()*, and a namedtuple object created by *burnman.tools.equilibration.get_equilibration_parameters()*. See documentation of these functions for the return types. If store_assemblage is True, each solution object also has an attribute called *assemblage*, which contains a copy of the input assemblage with the equilibrated properties. So, for a 2D grid of solution objects, one could call either sols[0][1].x[0] or sols[0][1].assemblage.pressure to get the pressure.

Return type

tuple

CHAPTER

SIXTEEN

THERMOBAROMETRY

16.1 Overview

One of the most common applications of thermodynamic modeling in geosciences is to estimate the pressure and temperature conditions at which a given mineral assemblage is stable. This process is known as thermobarometry. Burnman provides tools to perform thermobarometric calculations using a least-squares inversion approach, following and extending the Optimal Thermobarometry method described by [PH94].

The essential idea behind the method is to minimize a misfit scalar that is a function of reaction affinities, weighted by a covariance matrix that is a function of both measurement uncertainties and thermodynamic model uncertainties. The misfit scalar is defined as:

$$\Phi = \sum_i \sum_j a_i W_{ij} a_j$$
$$W_{ij} = (C_{ij} + M_{ij})^{-1}$$

where a_i is the affinity of reaction i, W_{ij} is the weighting matrix, C_{ij} is the covariance matrix of measurement uncertainties, and M_{ij} is the covariance matrix of thermodynamic model uncertainties.

The full documentation for the thermobarometry tools can be found at *Optimal thermobarometry*. The documentation for the Optimal Thermobarometry function is given below.

16.2 Implemented function

burnman.tools.thermobarometry.**estimate_conditions**(*assemblage*, *dataset_covariances=None*, *include_state_misfit=False*, *guessed_conditions=[1000000000.0, 1000.0]*, *pressure_bounds=[100000.0, 400000000000.0]*, *temperature_bounds=[300.0, 4000.0]*, *P_scaling=1000000.0*, *small_fraction_tol=0.0*, *max_it=100*)

Perform a least-squares inversion to find the optimal pressure and temperature for a given mineral assemblage. Algorithm modified from [PH94].

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to perform the inversion. Each solution phase in the assemblage must have its composition set along with its compositional covariance matrix (called *compositional_covariances*). If there are compositional degrees of freedom, they can be added by setting the attribute *free_compositional_vectors* on the relevant solution phases, where each row of the array corresponds to a free compositional vector, and the columns correspond to the amounts of endmembers of that phase in each vector.
- **dataset_covariances** (*dict, with keys 'endmember_names' and 'covariance_matrix'. Default is None, in which case only compositional uncertainties are considered.*) – The covariance data describing the uncertainties in endmember energies, taken from the thermodynamic dataset. For example, the covariance for the Holland et al. (2018) dataset can be obtained using `burnman.minerals.HGP_2018_ds633.cov()`.
- **include_state_misfit** (*bool*) – If True, includes the misfit from prior expectations on P and T. The assemblage must also have attributes *state_priors* and *state_inverse_covariances*.
- **guessed_conditions** (*np.array of shape (2,), optional, default None*) – Initial guess for pressure (Pa) and temperature (K). If not provided, the initial guess will be taken from the current state of the assemblage.
- **pressure_bounds** (*list of two floats*) – Bounds for pressure (Pa) during optimization.
- **temperature_bounds** (*list of two floats*) – Bounds for temperature (K) during optimization.
- **P_scaling** (*float*) – Scaling factor for pressure to improve numerical stability.
- **small_fraction_tol** (*float, optional, default 0.0*) – If > 0.0, reduces the number of endmembers in solution phases by transforming to a smaller set of independent endmembers using a greedy algorithm and excluding those with molar fractions smaller than this value during the inversion.
- **max_it** (*int, optional, default 100*) – Maximum number of iterations for the optimization algorithm.

Returns

Result object from the optimization containing the optimal conditions (x, which includes P, T, and any free compositional vectors) and other properties of the solution. These include the covariance matrix of the estimated parameters (xcov), the standard deviations of the estimated parameters (var), the correlation matrix (xcorr), the affinities of the independent reactions at the optimal conditions (affinities), the covariance matrix of the affinities (acov), the affinities weighted by the inverse square root of their covariance matrix (weighted_affinities), the partial derivatives of the affinities with respect to P and T (dadx), the number of independent reactions (n_reactions), the number of fitted parameters (n_params), the degrees of freedom (degrees_of_freedom), the chi-squared value (fun), the reduced chi-squared value (reduced_chisqr, given by the

reduced chi-squared divided by the number of degrees of freedom), and the fit quality (fit, given by the square root of the reduced chi-squared value).

Return type
OptimizeResult

CHAPTER

SEVENTEEN

TOOLS

Please see the API documentation for BurnMan's tools: *Tools*

CHAPTER

EIGHTEEN

UTILITY FUNCTIONS

BurnMan includes a number of utility functions that are used throughout the codebase. These functions are generally low-level and are not intended for direct use by the end user. Please see the API documentation for details: *Utilities*.

CHAPTER

NINETEEN

EXAMPLES

BurnMan comes with a large collection of example programs under examples/.

In this documentation section we include many of these examples. They are grouped here into three categories: *Class examples*, *Simple Examples* and *More Advanced Examples*. The *Class examples* introduce the main class types in BurnMan, and covers similar ground to *The BurnMan Tutorials* but in a little more detail. *Simple Examples* provides introductions to some of the seismic and chemical functions. *More Advanced Examples* covers functionality which is more suited to research projects.

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

19.1 Class examples

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

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

19.1.1 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 for Minerals*).

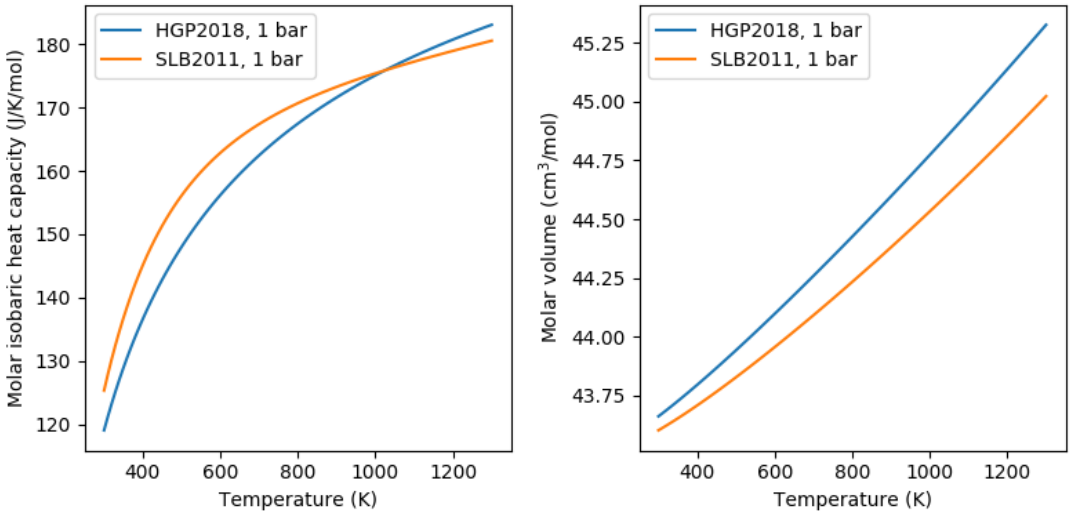
Uses:

- *Mineral Datasets*
- `burnman.Mineral`

Demonstrates:

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

Resulting figure:



19.1.2 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))

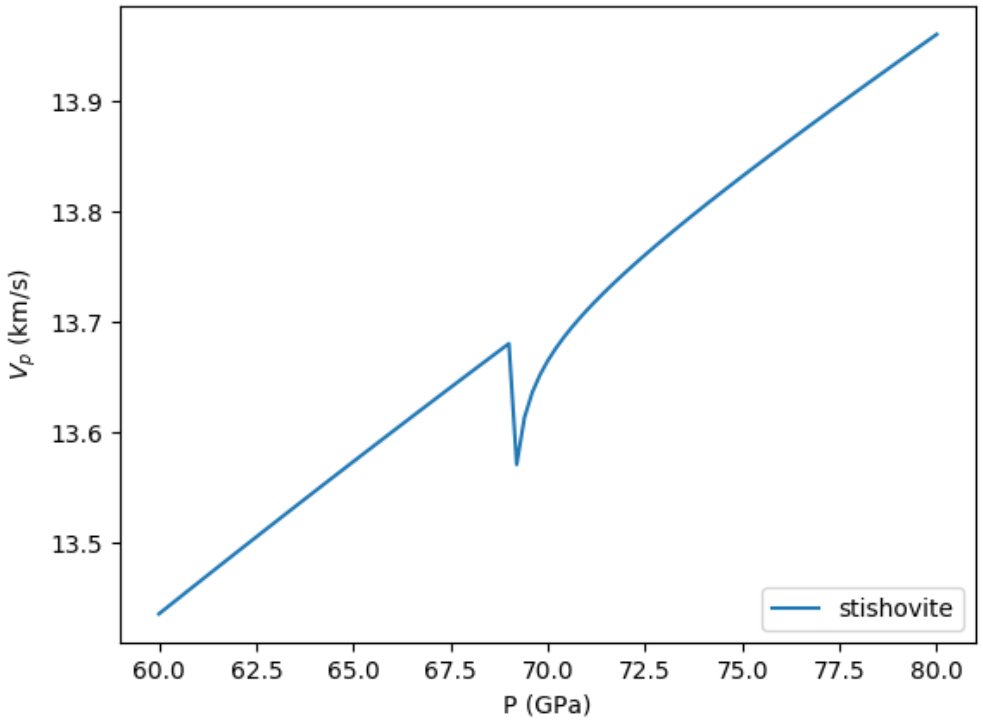
Uses:

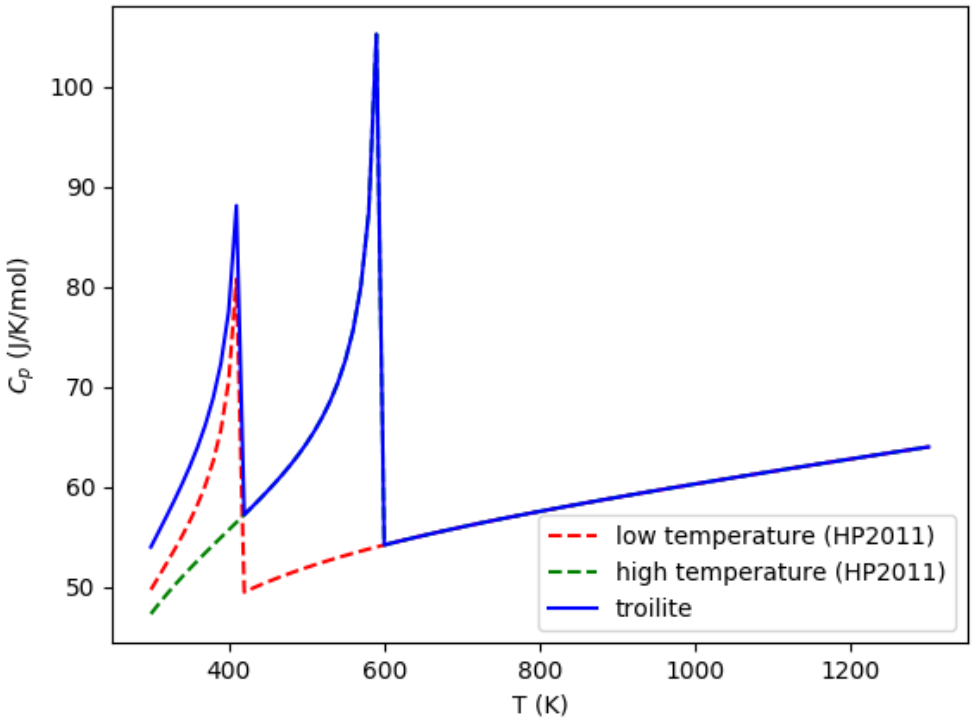
- *Mineral Datasets*

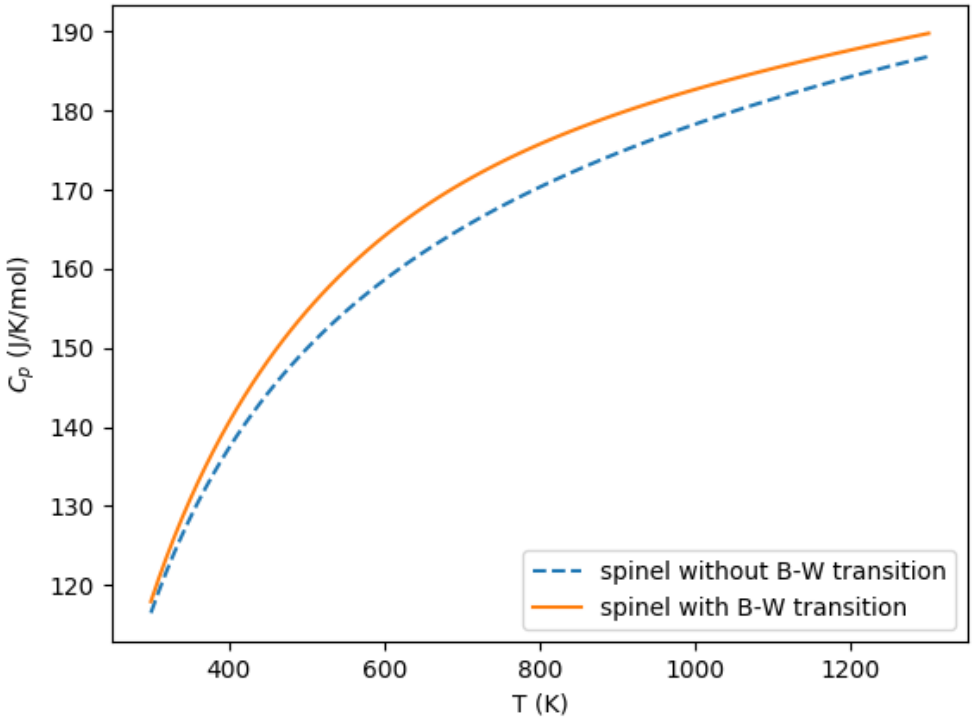
Demonstrates:

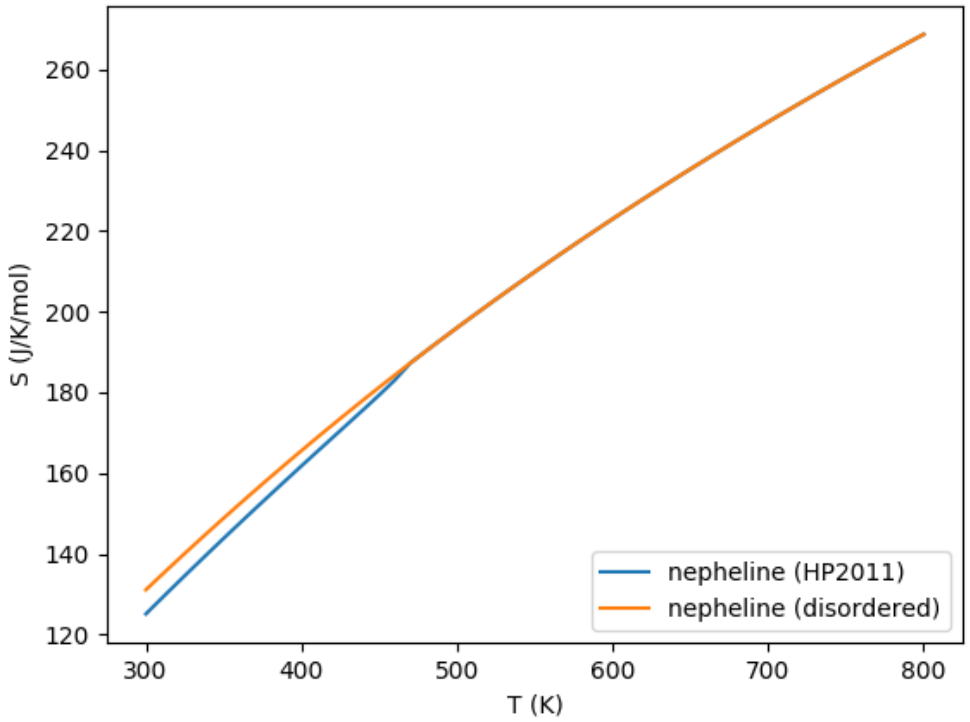
- creating a mineral with excess contributions
- calculating thermodynamic properties

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



19.1.3 example_solution

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

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

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

These solutions can potentially deal with:

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

Uses:

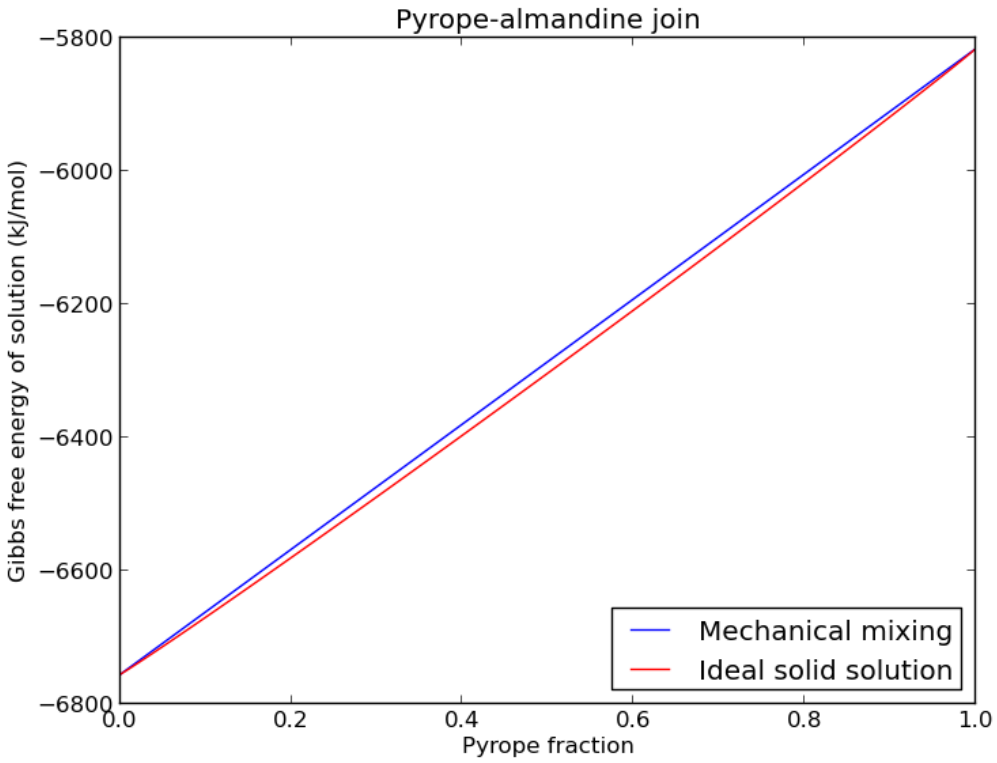
- *Mineral Datasets*
- `burnman.Solution`

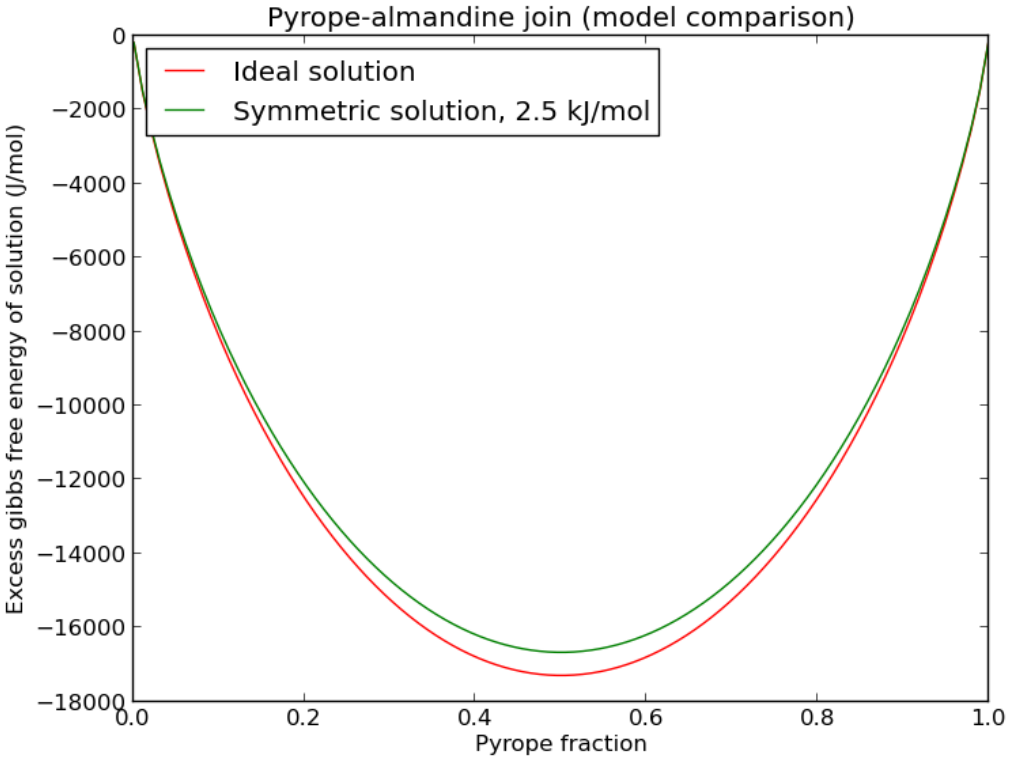
- *burnman.SolutionModel*

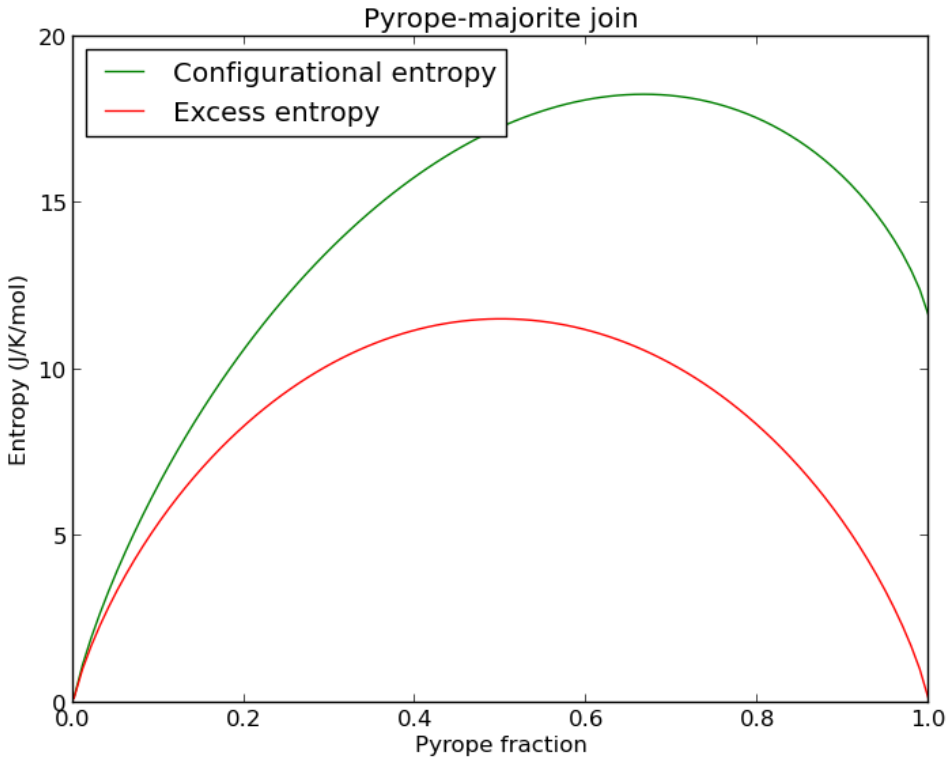
Demonstrates:

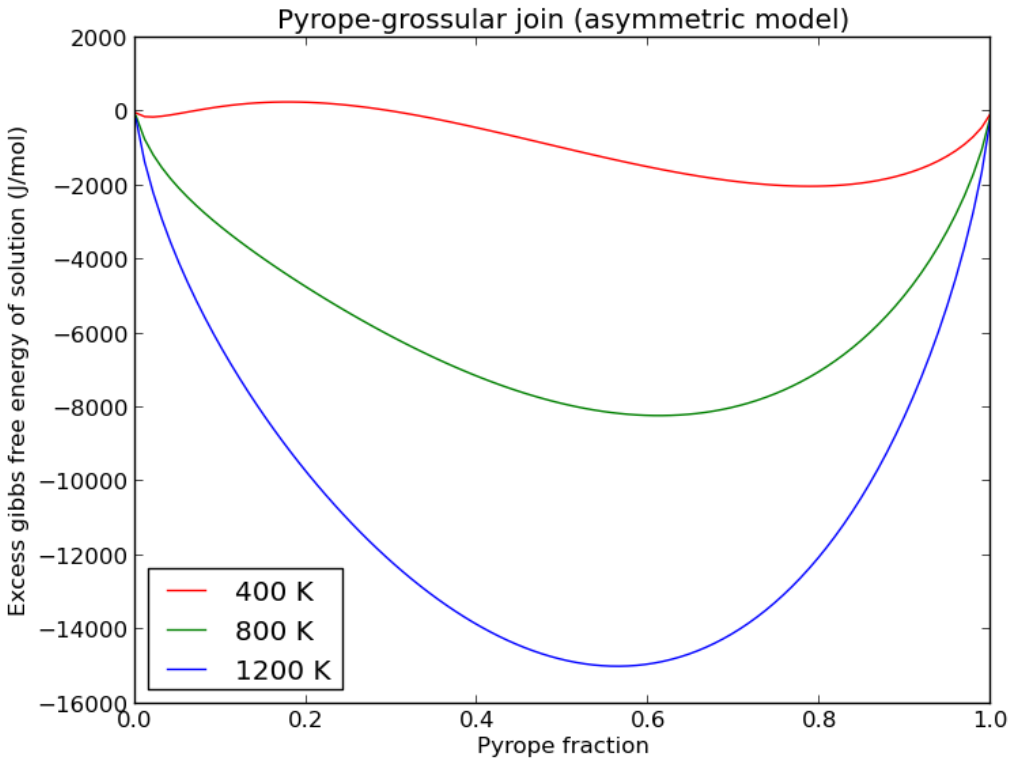
- Different ways to define a solution
- How to set composition and state
- How to output thermodynamic and thermoelastic properties

Resulting figures:




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


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



19.1.4 example_composite

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

Uses:

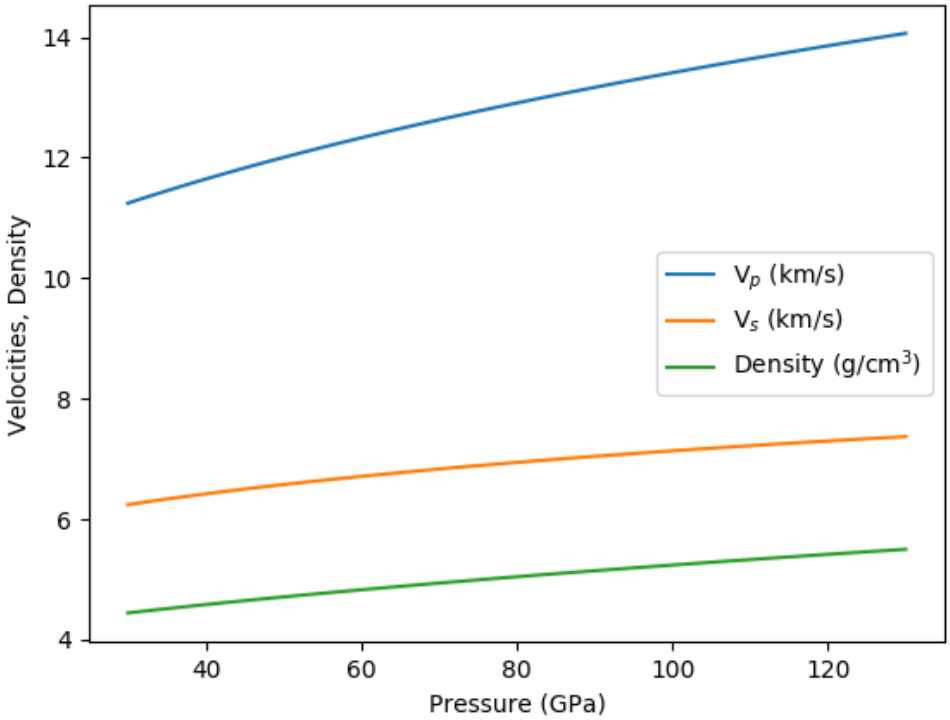
- *Mineral Datasets*
- `burnman.Mineral`
- `burnman.Solution`
- `burnman.Composite`

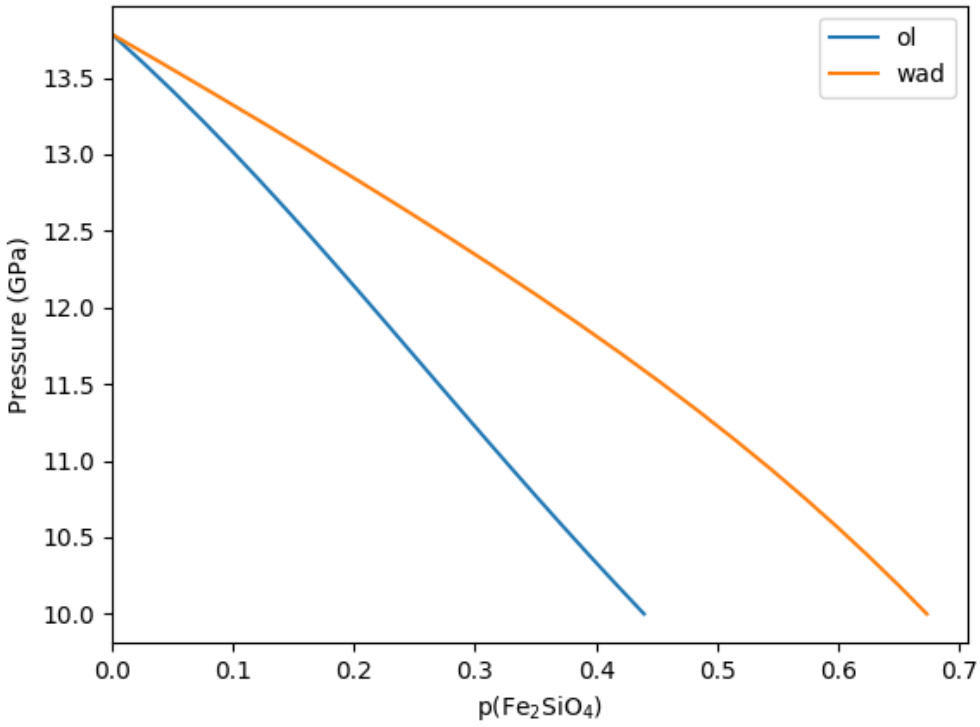
Demonstrates:

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

Please see example_equilibrate.py for more advanced (and user-friendly) examples of equilibration of a composite material.

Resulting figures:





19.1.5 example_calibrants

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

Uses:

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

Demonstrates:

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

19.1.6 example_anisotropy

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

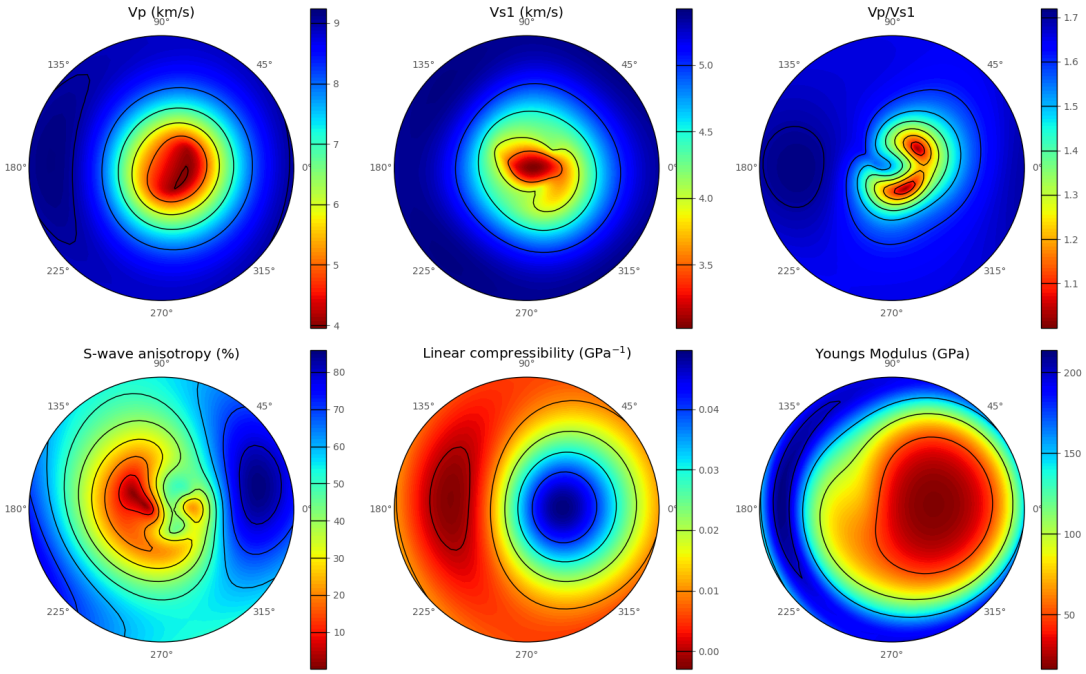
Specifically uses:

- *burnman.AnisotropicMaterial*

Demonstrates:

- anisotropic functions

Resulting figure:



19.1.7 example_anisotropic_mineral

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

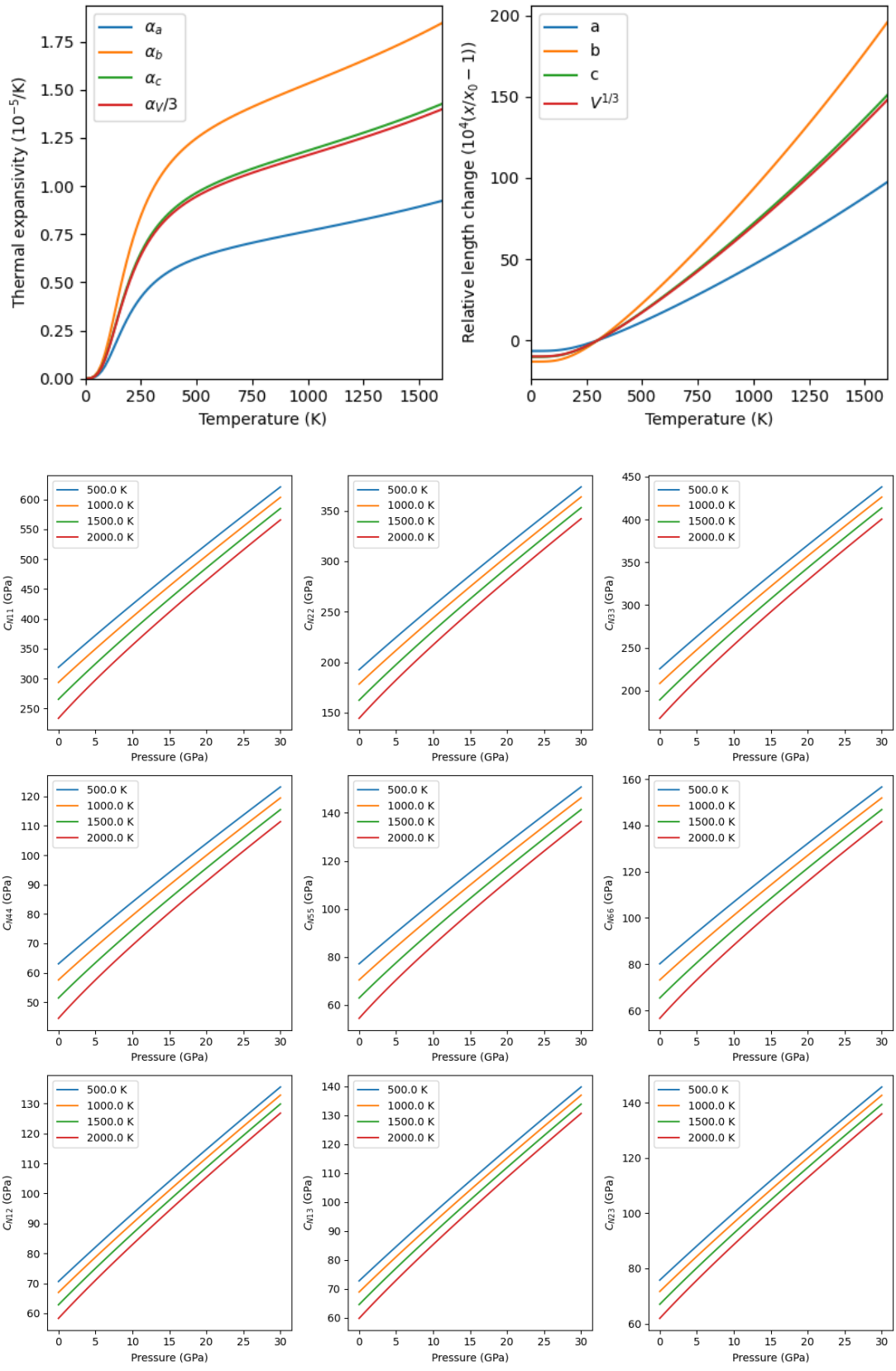
Specifically uses:

- *burnman.AnisotropicMineral*

Demonstrates:

- anisotropic functions

Resulting figure:


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

19.1.8 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. Stacey, 1977 continental [Sta77]
4. Stacey, 1977 oceanic [Sta77]
5. Custom tabulated geotherm
6. Adiabatic geotherm for a given mineral assemblage

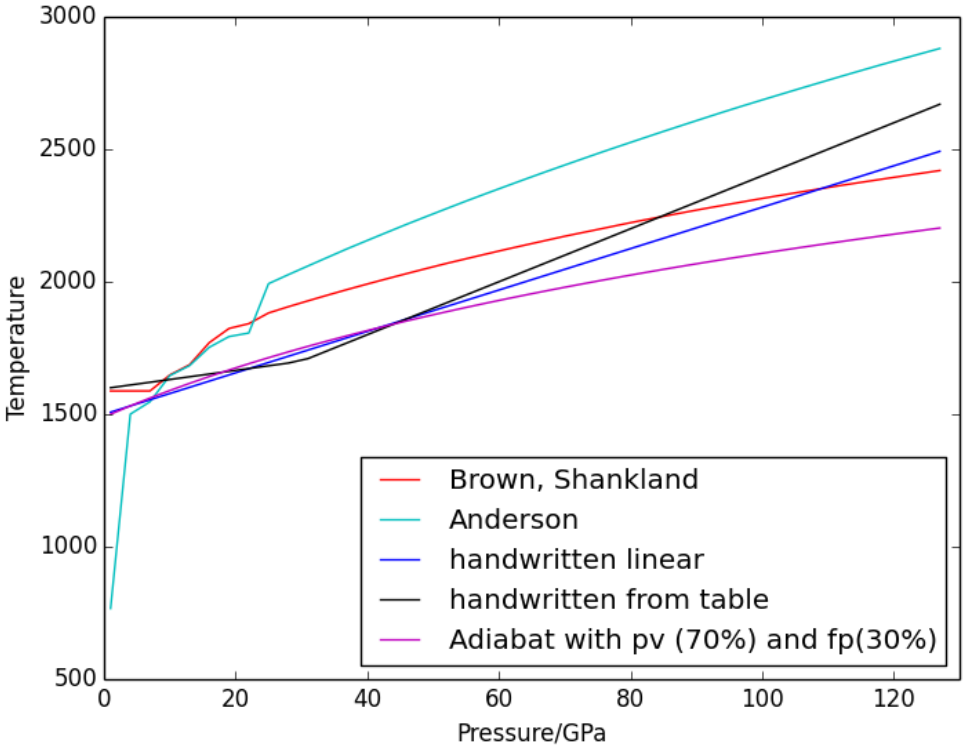
Uses:

- `burnman.geotherm.BrownShankland`
- `burnman.geotherm.Anderson`
- `burnman.geotherm.StaceyContinental`
- `burnman.geotherm.StaceyOceanic`
- `burnman.geotherm.Geotherm`
- `burnman.geotherm.AdiabaticGeotherm`
- *`burnman.Composite`* for adiabat

Demonstrates:

- the available geotherms

Resulting figure:



19.1.9 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

19.2 Simple Examples

The following is a list of simple examples:

- *example_beginner*,
- *example_seismic*,

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

19.2.1 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.

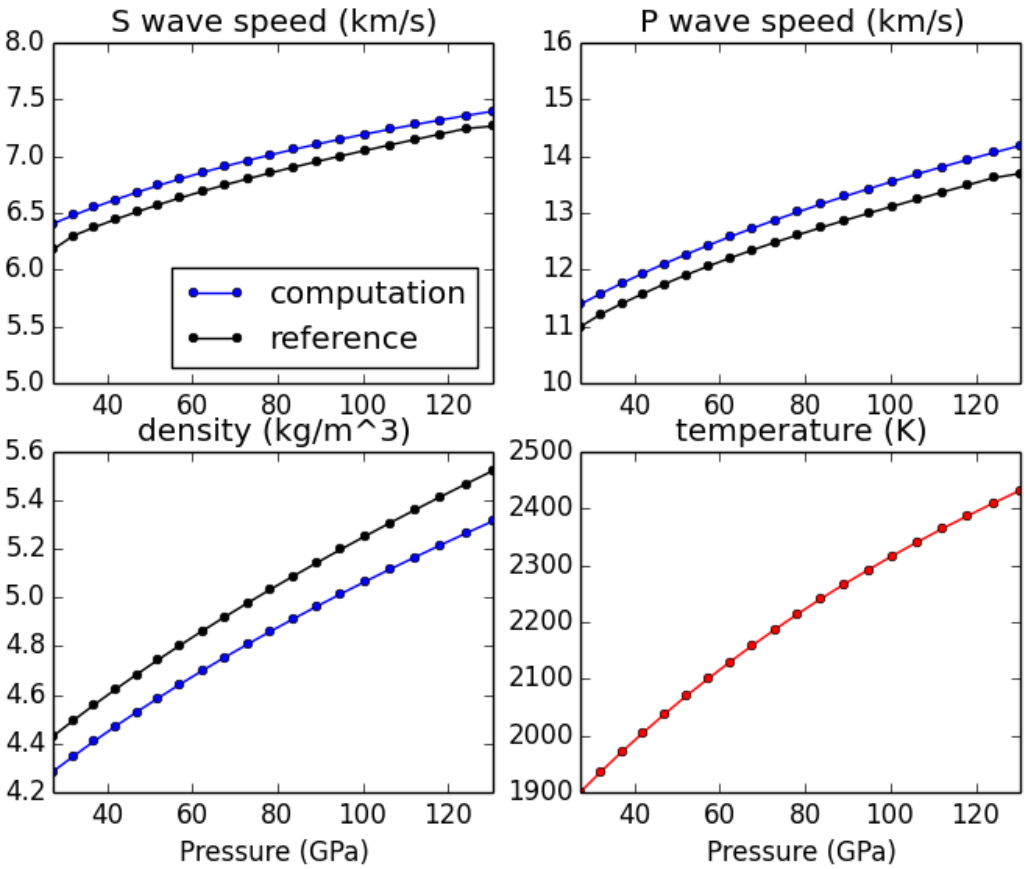
Uses:

- *Mineral Datasets*
- *burnman.Composite*
- burnman.seismic.PREM
- burnman.geotherm.BrownShankland
- *burnman.Material.evaluate()*

Demonstrates:

- creating basic composites
- calculating thermoelastic properties
- seismic comparison

Resulting figure:



19.2.2 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:

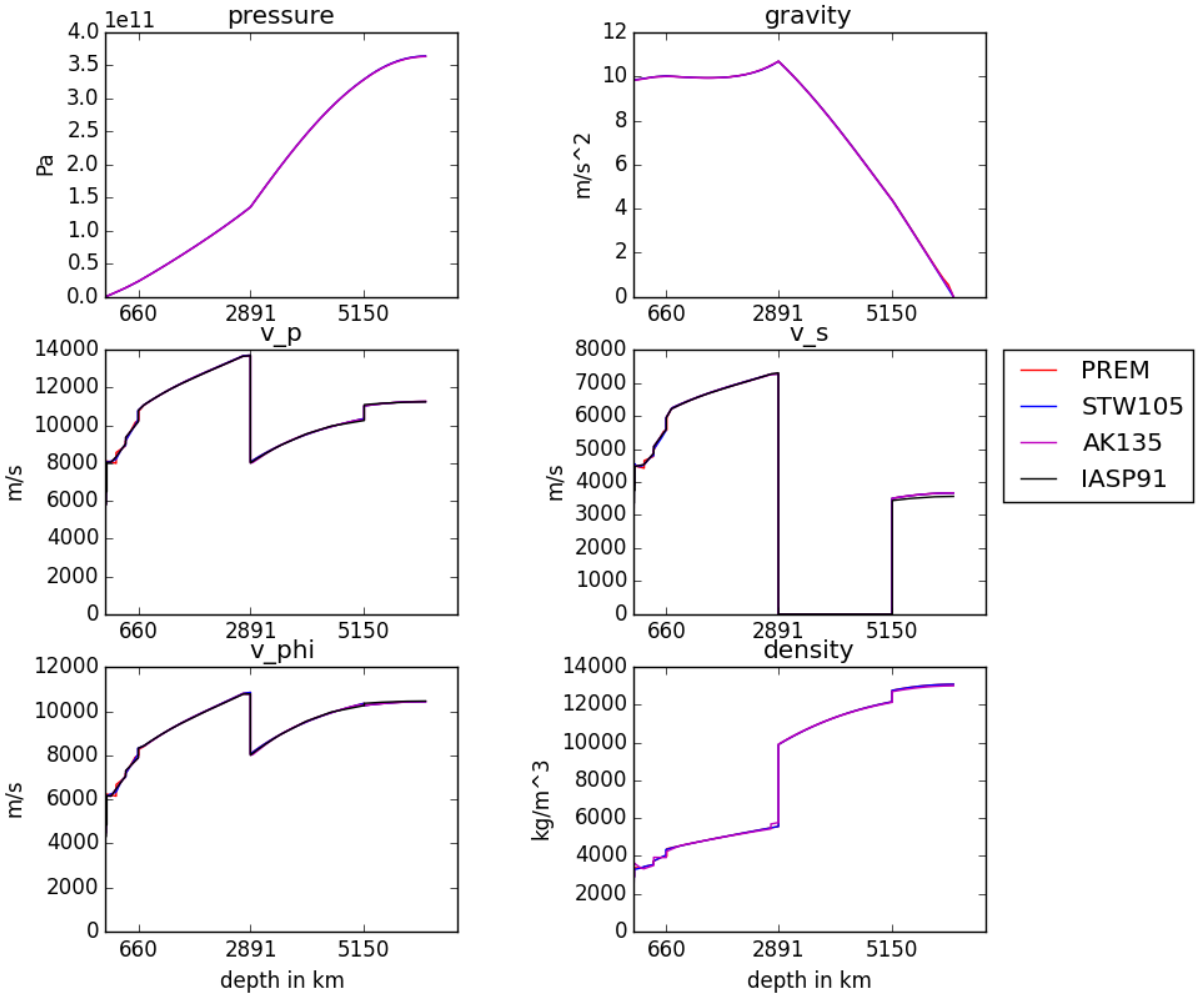
- *Seismic Models*

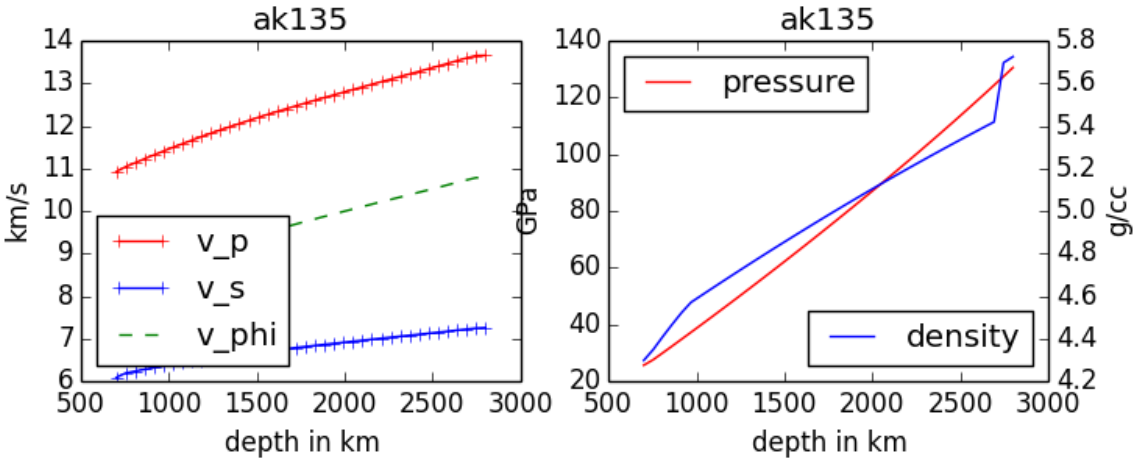
Demonstrates:

- Utilization of library seismic models within BurnMan

- Input of user-defined seismic models

Resulting figures:





19.2.3 example_composite_seismic_velocities

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

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

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

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

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

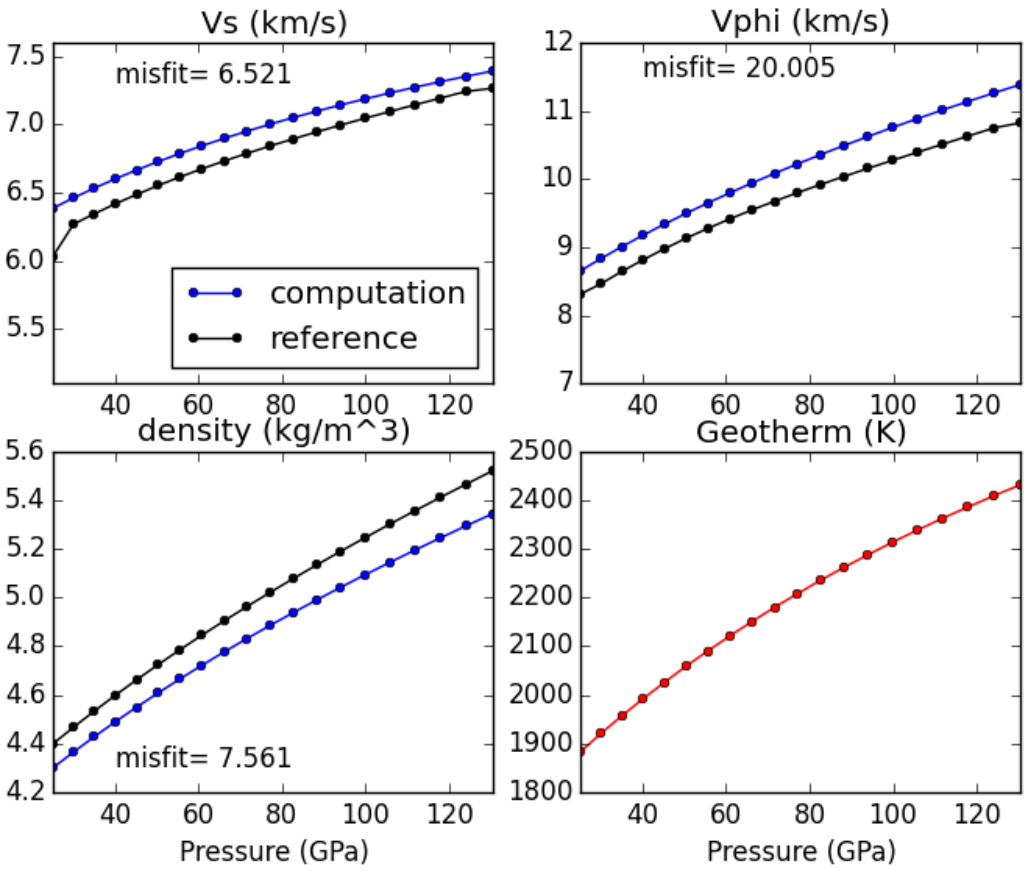
Uses:

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

Demonstrates:

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

Resulting figure:



19.2.4 example_averaging

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

1. Voigt-Reuss-Hill

2. Voigt averaging
3. Reuss averaging
4. Hashin-Shtrikman averaging

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

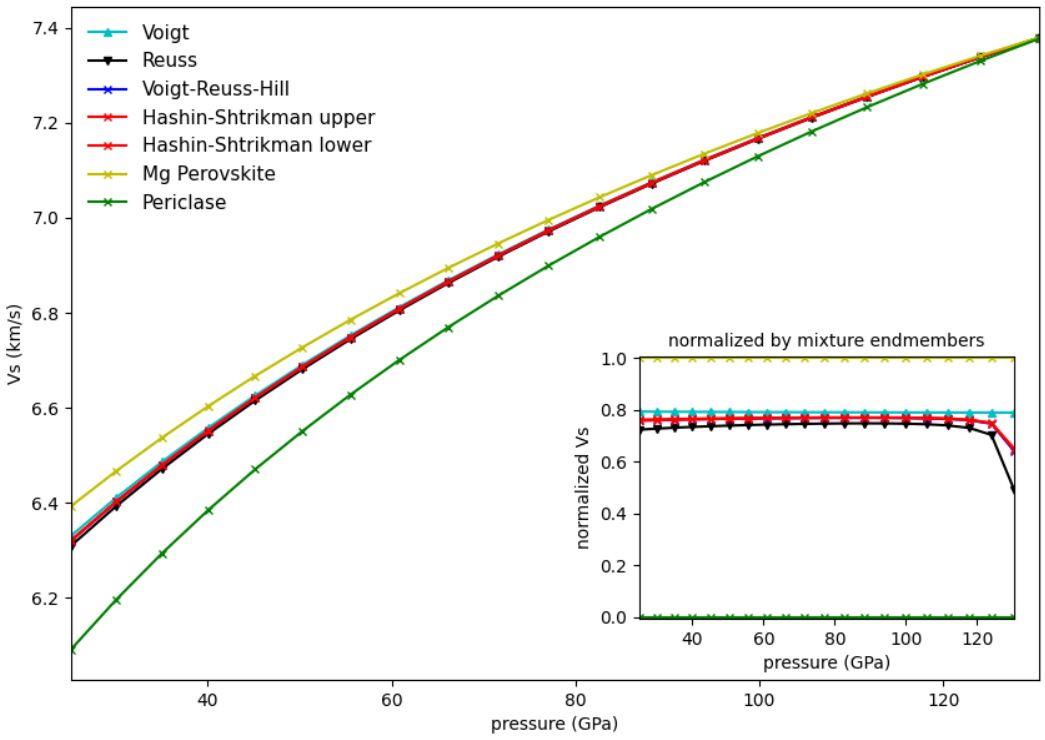
Specifically uses:

- *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:



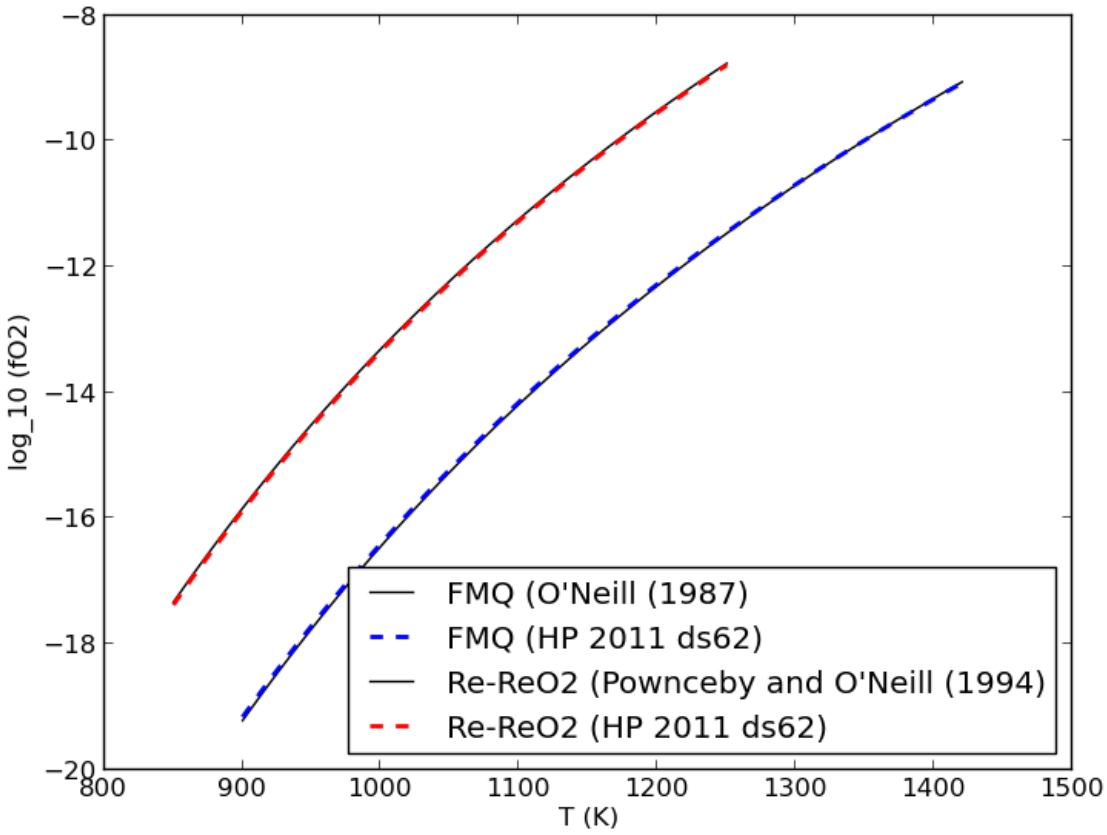
19.2.5 example_chemical_potentials

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

Demonstrates:

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

Resulting figure:



19.3 More Advanced Examples

Advanced examples:

- *example_hyperelastic_stress_rates*,
- *example_spintransition*,
- *example_spintransition_thermal*,
- *example_user_input_material*,
- *example_optimize_pv*,

- *example_compare_all_methods*,
- *example_build_planet*,
- *example_fit_composition*,
- *example_fit_data*,
- *example_fit_eos*,
- *example_fit_solution*,
- example_optimal_thermobarometry,
- *example_equilibrate*, and
- *example_olivine_binary*.

19.3.1 example_hyperelastic_stress_rates

In the Earth, tectonic and other geodynamic processes cause large-scale movements of rocks. This movement can be seen as a combination of translation, rotation, and deformation.

The most natural way to describe the deformation of a local rock element is to track the deformation gradient F, which maps the initial configuration of the rock element to its current configuration. This removes the influence of rigid translation and includes both rotation and deformation. In BurnMan, we can compute the thermodynamic and elastic response to any applied elastic deformation gradient F. This includes the internal energy, Cauchy stress, and elastic stiffnesses, in addition to the more directly calculated quantities of volume change and finite strain.

Geodynamic simulations usually process the physical response in a different way. Instead of evaluating stress directly from the current state, they evolve stress incrementally using the velocity gradient and the stress retained from the previous time step. This approach fits naturally into a calculation that includes viscous relaxation and plastic flow. However, the retained stress must be transported with the moving rock. In particular, rotation changes its components in fixed axes, even if no further stretching occurs. An update based only on the rate of stretching would miss this change.

The Truesdell, Green-Naghdi and Zaremba-Jaumann rates provide different ways to express evolving stress in the same elastic material, taking into account changes in configuration. The rates themselves are exact definitions; the integration schemes used here are first-order approximations. They differ in how they transport the stress and how they process the constitutive response. Their integration over finite time steps can give different answers, raising the questions: How large are the accumulated errors, and how do they depend on the number of steps?

BurnMan's hyperelastic equation of state provides a reference for answering these questions. It gives the exact nonlinear constitutive response of the chosen material model, up to numerical differentiation and roundoff errors. Stress can be evaluated directly at each prescribed deformation gradient F, without accumulating errors through incremental updates.

In this example we run two experiments, comparing the hyperelastic Cauchy stress with the stress derived from the three objective rates. The first experiment rotates a prestretched crystal through 90 degrees. Its principal stresses remain constant, while its stress components rotate. All three objective updates reproduce the BurnMan result. Omitting rotation leaves the stress pointing in the original direction.

The second experiment combines shear, dilation and rotation at constant temperature. The plots compare the accumulated stresses with BurnMan's direct calculation, then show how the errors decrease as the steps become smaller. All three converge to the same hyperelastic response in the limit of small time steps.

To implement the comparison, we define L = Fdot inv(F), D = sym(L), W = skew(L), and Omega = Rdot R.T, where F = R U is the polar decomposition of the total deformation gradient. The Cauchy-stress rates are given below; sigma_dot follows the moving rock element:

```
Zaremba-Jaumann stress rate: sigma_dot - W sigma + sigma W
Green-Naghdi stress rate:    sigma_dot - Omega sigma + sigma Omega
Truesdell stress rate:       sigma_dot - L sigma - sigma L.T + tr(D) sigma
```

BurnMan provides the current isothermal elastic stiffness B, which relates the rate of stretching D to the Jaumann stress rate at constant temperature. Their contraction is written B:D, with (B:D)_ij = sum_kl B_ijkl D_kl. The full rate of change of the Cauchy stress in fixed axes is:

```
sigma_dot = B:D + W sigma - sigma W
```

Moving the rotation terms to the left gives the Jaumann rate listed above. Green-Naghdi subtracts rotation based on Omega instead of W. To keep the same sigma_dot, the difference between these rotation terms must remain on the right. Truesdell also subtracts terms involving stretching and volume change, which must likewise be accounted for on the right. Rearranging the same stress evolution equation therefore gives:

```
Jaumann rate      = B:D
Green-Naghdi rate = B:D + (W - Omega) sigma - sigma (W - Omega)
Truesdell rate    = B:D - D sigma - sigma D + tr(D) sigma
```

References:

- Beuchert and Podladchikov (2010): Viscoelastic mantle convection and lithospheric stresses https://academic.oup.com/gji/article/183/1/35/594736
- Wallace (1967): Thermoelasticity of Stressed Materials and Comparison of Various Elastic Constants; see Eq. 2.35 for the definition of BurnMan's stress–strain tangent B. https://doi.org/10.1103/PhysRev.162.776
- Objective stress rates: https://en.wikipedia.org/wiki/Objective_stress_rate https://mooseframework.inl.gov/source/materials/lagrangian/ComputeLagrangianObjectiveStress.html

Uses:

- `burnman.HyperelasticMineral`

Demonstrates:

- Rotation of an initially non-hydrostatic stress.
- Conversion of the constitutive response between objective stress rates.
- Numerical convergence against a hyperelastic reference.

`examples.example_hyperelastic_stress_rates.`**`rotation_z`**`(`*`angle`*`)`

Construct an active rotation about the z axis.

Parameters
angle – Rotation angle in radians, positive from x towards y.

Returns
A (3, 3) numpy array acting on column vectors.

examples.example_hyperelastic_stress_rates.**kinematics**(*t*, *shear*, *rotation*, *log_volume*)

Evaluate shear, isotropic dilation and rotation along a prescribed path.

All rates are derivatives with respect to the dimensionless load parameter t, rather than physical time.

Parameters

- **t** – Load parameter in [0, 1].
- **shear** – Final dimensionless simple shear, with x displacement proportional to the initial y coordinate.
- **rotation** – Final superposed rotation about z, in radians.
- **log_volume** – Final logarithmic volume ratio, ln(V_final / V_initial).

Returns
Dictionary of (3, 3) numpy arrays: deformation gradient `F`, stretching rate `D`, spin `W`, polar-rotation rate `Omega`, polar rotation `Green-Naghdi`, and integrated-spin rotation `Zaremba-Jaumann`. Both rotation matrices start at the identity.

examples.example_hyperelastic_stress_rates.**sample_path**(*mineral*, *cell_vectors*, *temperature*, *n_steps*, *shear*, *rotation*, *log_volume*)

Sample the hyperelastic stress and loading (B:D) along a simple path in the space of deformation gradients.

The mineral is left at the final sampled state.

Parameters

- **mineral** – HyperelasticMineral with output_frame set to `current`.
- **cell_vectors** – Initial molar cell vectors as the rows of a (3, 3) array, with determinant equal to the molar volume in m^3/mol.
- **temperature** – Constant temperature in K.
- **n_steps** – Positive integer number of intervals; both endpoints are included, giving n_steps + 1 samples.
- **shear** – Final dimensionless simple shear in the x direction with gradient in y.
- **rotation** – Final superposed rotation about z, in radians.
- **log_volume** – Final logarithmic volume ratio, ln(V_final / V_initial).

Returns
Dictionary with `times` (dimensionless load parameters, shape (n_steps + 1,)), `states` (kinematics dictionaries), and `stress` and `loading` arrays of shape (n_steps + 1, 3,

3). Stress is Cauchy stress in Pa; loading is B:D in Pa per unit load parameter. Both are expressed in the fixed spatial axes.

examples.example_hyperelastic_stress_rates.**integrate_rates**(*data*, *stride=1*, *consistent=True*)

Integrate each stress-rate equation using a first-order update.

Only the initial EoS stress is used to initialise the stress histories. Later stresses are accumulated using the rate equations, with B:D sampled from the EoS at the start of each update. The right-hand side of each stress-rate equation is evaluated at the start of the step.

Parameters

- **data** – Dictionary returned by sample_path.
- **stride** – Positive integer number of sampled intervals per update. Must divide the sampled interval count; 1 uses every sample.
- **consistent** – If True, add the stress-dependent terms required when rewriting the Jaumann equation in Green-Naghdi or Truesdell form, so all three equations describe the same stress evolution. B:D alone is the right-hand side of the Jaumann equation. If False, deliberately omit these terms and set the right-hand side of all three equations to B:D.

Returns

Dictionary keyed by the three names in RATES and `No rotation`. Each value is a Cauchy-stress history in Pa, with shape (n_updates + 1, 3, 3), where n_updates is the sampled interval count divided by stride. `No rotation` adds B:D without transporting stress.

examples.example_hyperelastic_stress_rates.**relative_error**(*predicted*, *reference*)

Return the maximum Frobenius error divided by the largest reference norm.

Parameters

- **predicted** – Predicted stress history, a numpy array of shape (n_samples, 3, 3).
- **reference** – Reference history at the same samples, with the same shape and units. Its maximum Frobenius norm must be nonzero.

Returns

Dimensionless scalar relative error, as a fraction, not a percentage. Each maximum is taken over all samples.

examples.example_hyperelastic_stress_rates.**rotation_experiment**(*mineral*, *cell_vectors*, *temperature*, *n_steps*)

Rotate a prestretched crystal and compare stress components in fixed axes.

Apply a volume-preserving prestretch, then rotate through 90 degrees. Print the maximum stress errors and leave the mineral at the final state.

Parameters

- **mineral** – HyperelasticMineral with output_frame set to `current`.

- **cell_vectors** – Initial molar cell vectors as rows of a (3, 3) array, before pre-stretching; its determinant is the molar volume in m^3/mol.
- **temperature** – Constant temperature in K.
- **n_steps** – Positive integer number of rotation-update intervals.

Returns

A matplotlib.figure.Figure comparing the normal and shear stresses with the hyperelastic reference, including an update that omits rotation.

examples.example_hyperelastic_stress_rates.**deformation_experiment**(*mineral*, *cell_vectors*, *temperature*, *n_steps*)

Conduct a convergence study of three objective stress rates. The study runs the same deformation path with different numbers of stress updates, then compares the accumulated stresses with the hyperelastic reference.

The deformation path combines shear, dilation and rotation at constant temperature. Errors and convergence orders are calculated, printed to stdout and plotted. The mineral is left at the final state.

Parameters

- **mineral** – HyperelasticMineral with output_frame set to `current`.
- **cell_vectors** – Initial molar cell vectors as rows of a (3, 3) array, with determinant equal to the molar volume in m^3/mol.
- **temperature** – Constant temperature in K.
- **n_steps** – Positive integer number of updates in the stress plots. The convergence study uses n_steps, 2*n_steps, 4*n_steps and 8*n_steps.

Returns

A matplotlib.figure.Figure showing stress histories, polar rotation versus integrated spin, and stress-error convergence.

19.3.2 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.

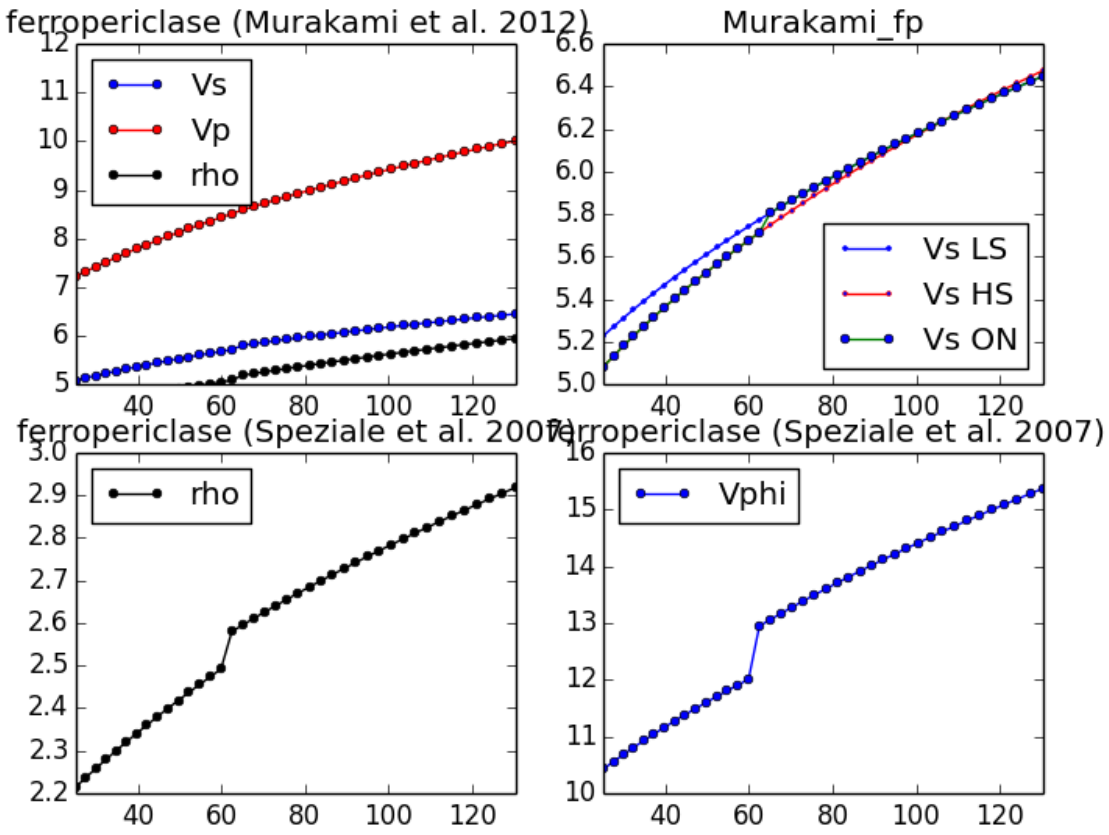
Specifically uses:

- `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:



19.3.3 example_spintransition_thermal

This example illustrates how to create a non-ideal solution model for (Mg,Fe^{HS},Fe^{LS})O ferropericlase that has a gradual spin transition at finite temperature.

In the first part of the example, we create a Solution class from scratch that incorporates three endmembers: MgO, FeO in the high spin state, and FeO in the low spin state. We then embed this solution in a Relaxed-Solution class that allows us to define the isochemical high spin to low spin transition as a free relaxation vector that is allowed to vary freely to minimize the Gibbs free energy.

In the second part of the example, we illustrate how to use the ferropericlase solution model directly from the Stixrude and Lithgow-Bertelloni (2024) database, again embedding it in a RelaxedSolution to incorporate the spin transition. We then show how to calculate thermodynamic and elastic properties that incorporate the effects of the spin transition over a range of pressures and temperatures along an oceanic geotherm.

In this example, we 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 implemented mixing model explicitly precludes long range order.

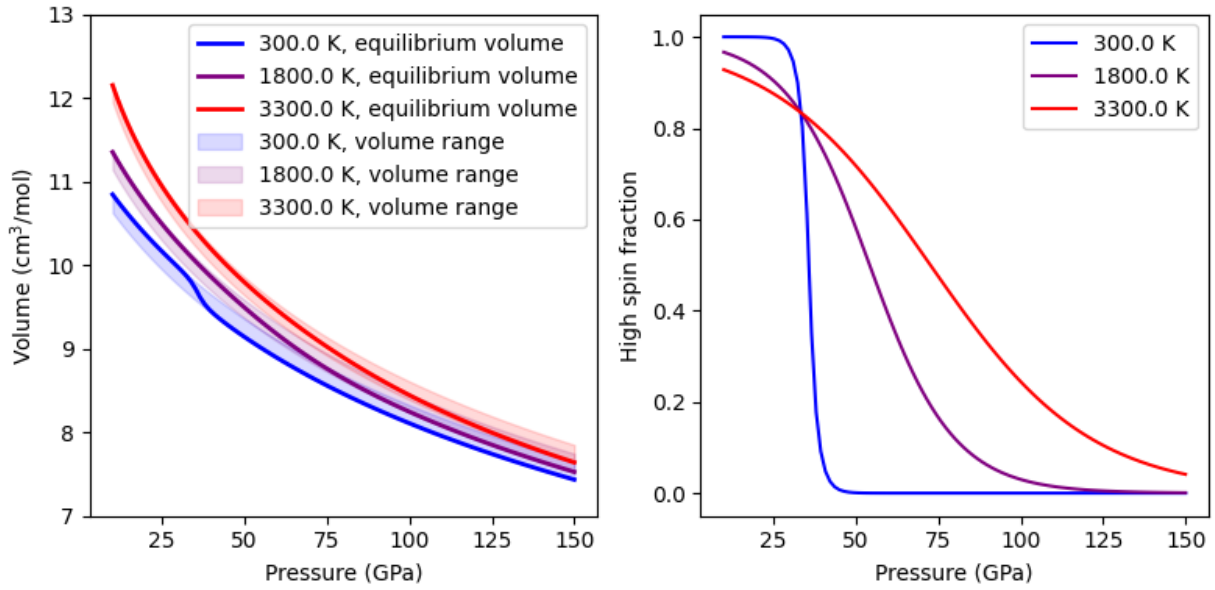
Specifically uses:

- *burnman.Mineral*
- *burnman.Solution*
- *burnman.RelaxedSolution*

Demonstrates:

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

Resulting figure:



19.3.4 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

19.3.5 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:

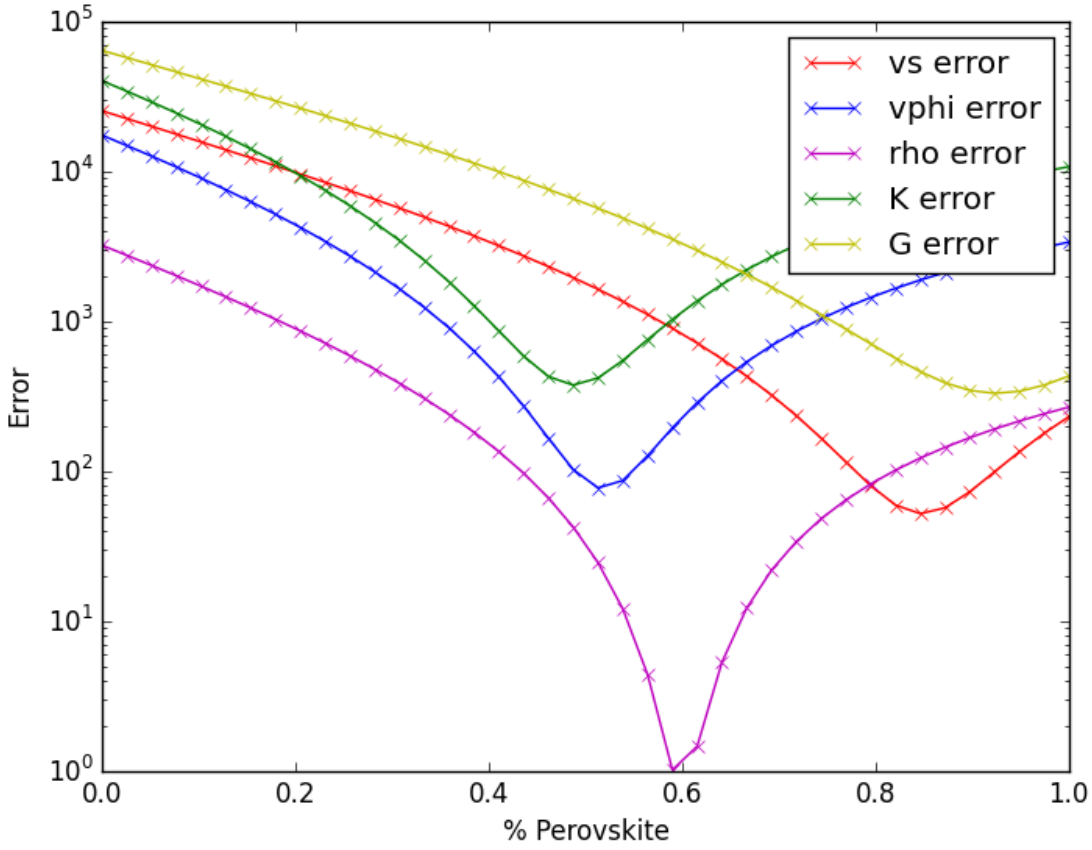
- *Mineral Datasets*

- *burnman.Composite*
- burnman.seismic.PREM
- burnman.geotherm.BrownShankland
- *burnman.Material.evaluate()*
- *burnman.utils.math.l2_norm_profiles()*

Demonstrates:

- compare errors between models
- loops over models

Resulting figure:



19.3.6 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.

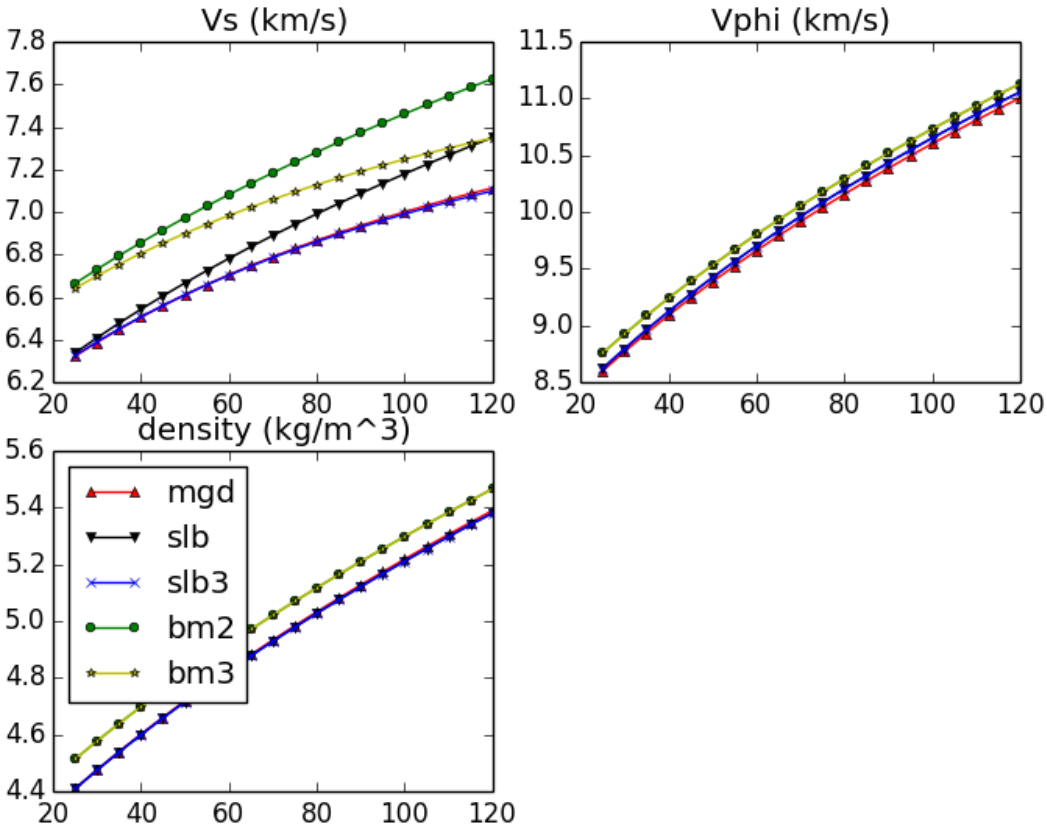
Specifically uses:

- *Equations of State for Minerals*

Demonstrates:

- Each method for calculating velocity profiles currently included within BurnMan

Resulting figure:



19.3.7 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.

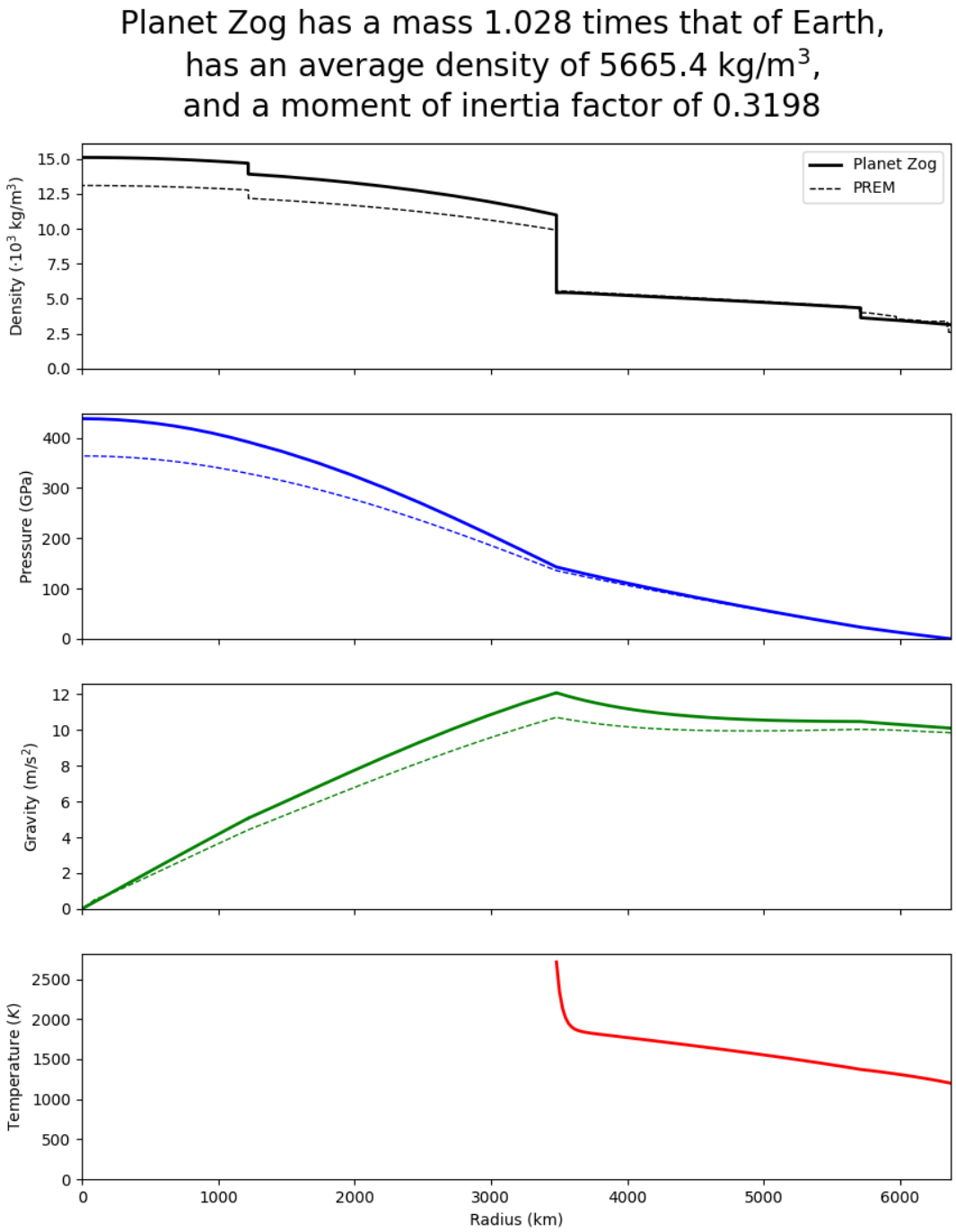
Uses:

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

Demonstrates:

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

Resulting figure:


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

19.3.8 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.

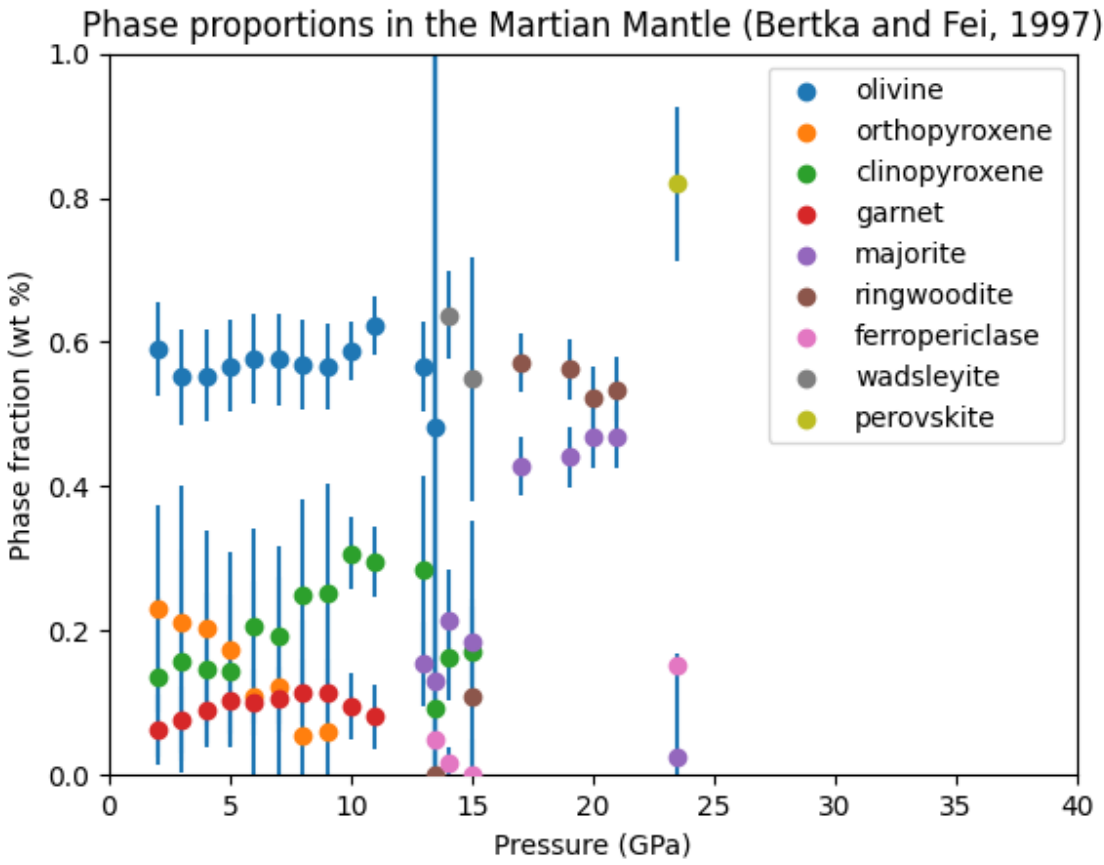
Uses:

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

Demonstrates:

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

Resulting figure:



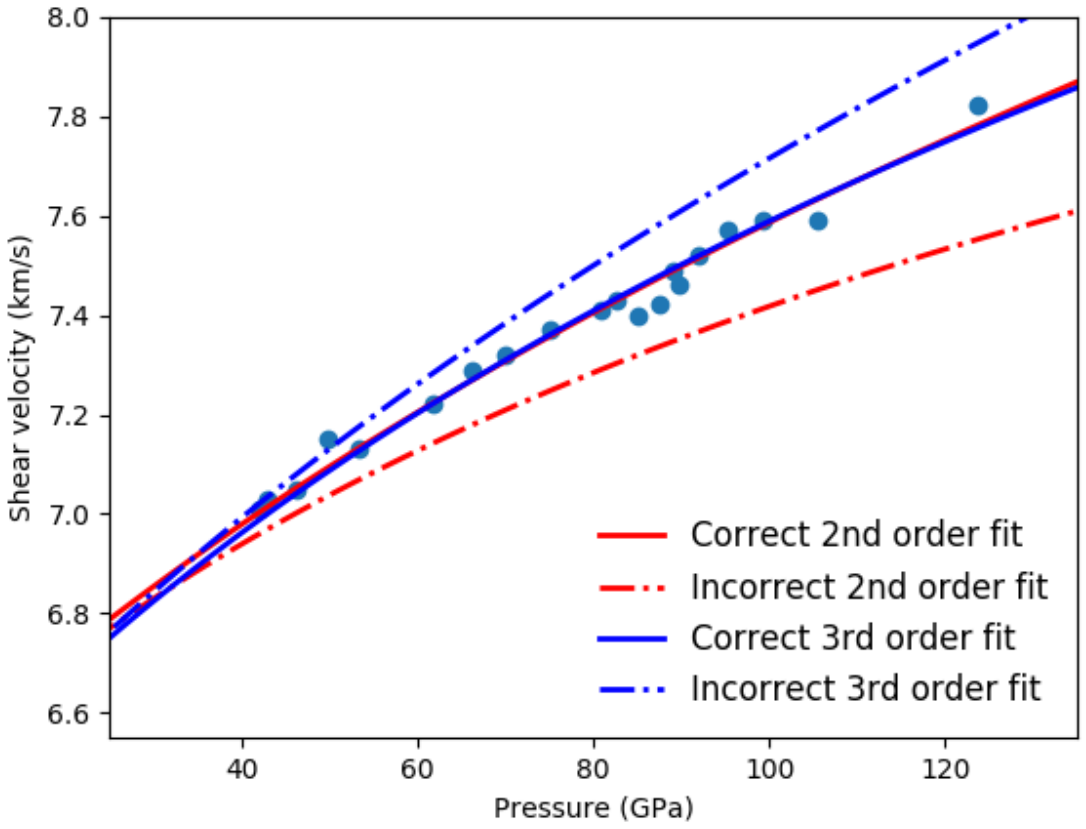
19.3.9 example_fit_data

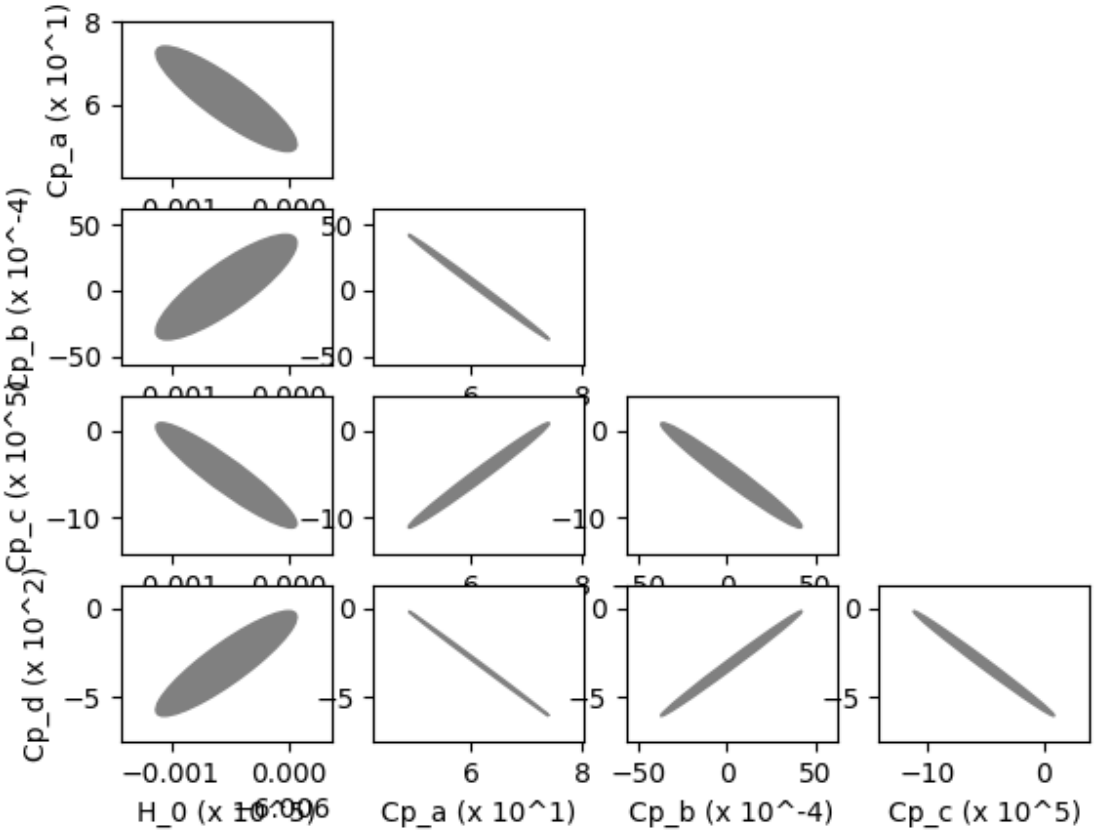
This example demonstrates BurnMan's functionality to fit various mineral physics data to an EoS of the user's choice.

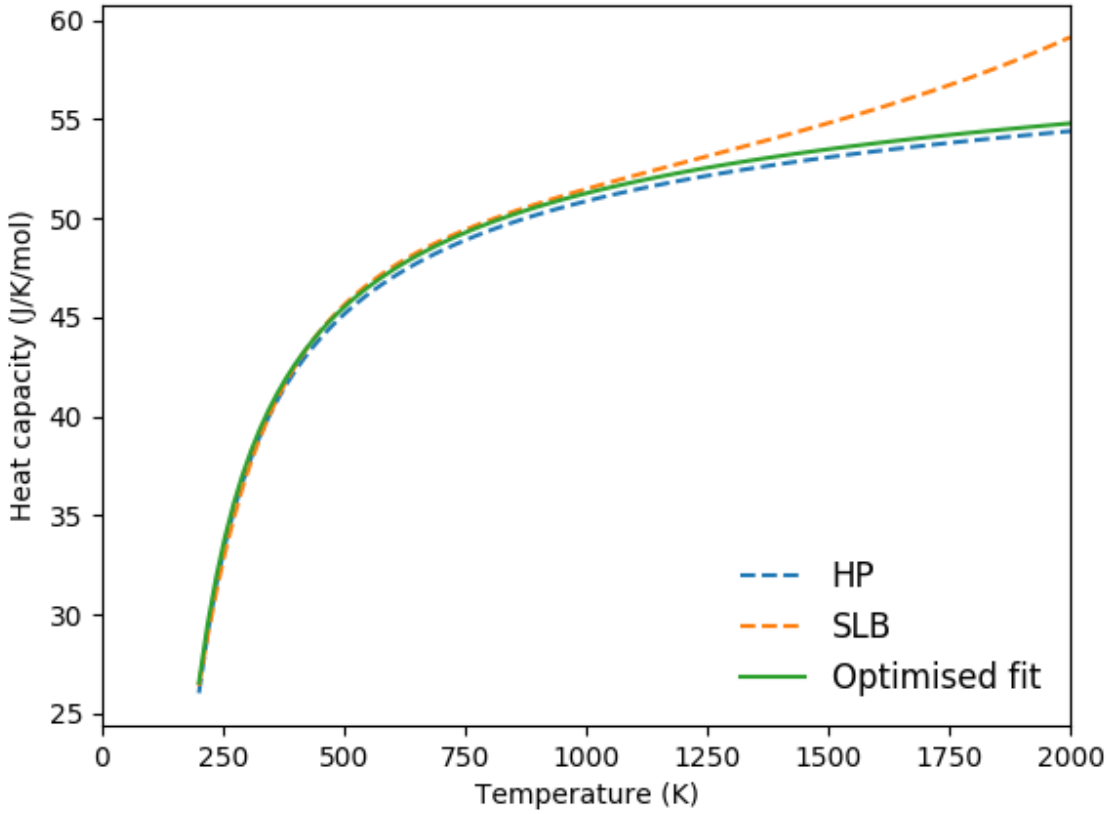
Please note also the separate file example_fit_eos.py, which can be viewed as a more advanced example in the same general field.

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)



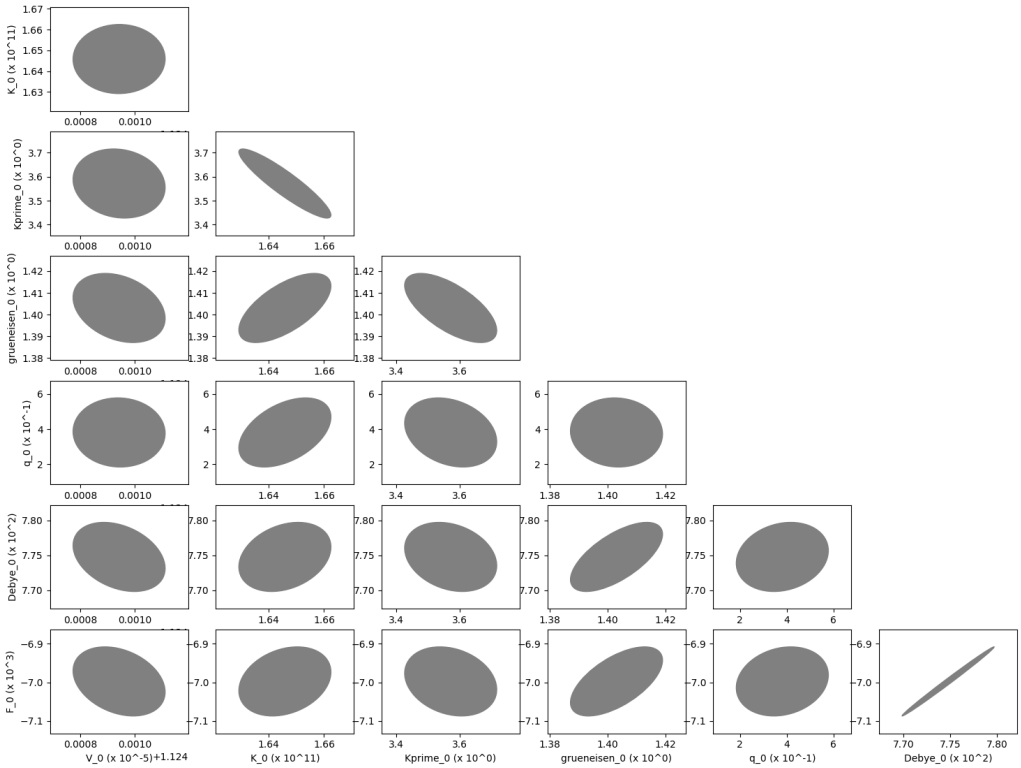
19.3.10 example_fit_eos

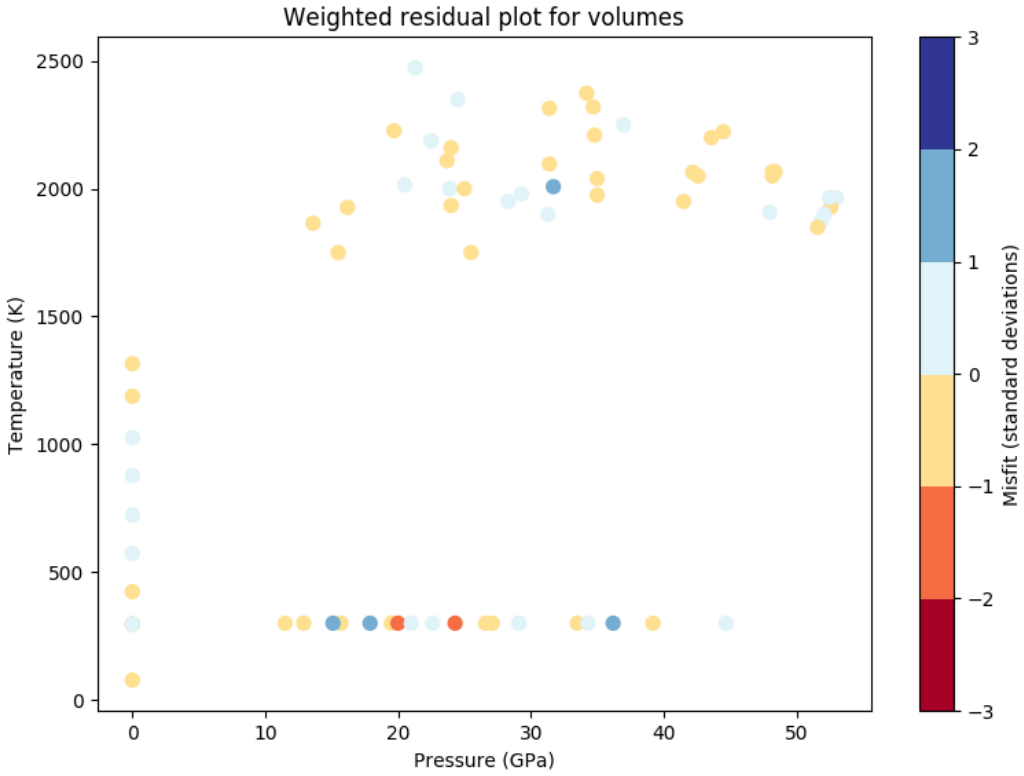
This example demonstrates BurnMan's functionality to fit data to an EoS of the user's choice.

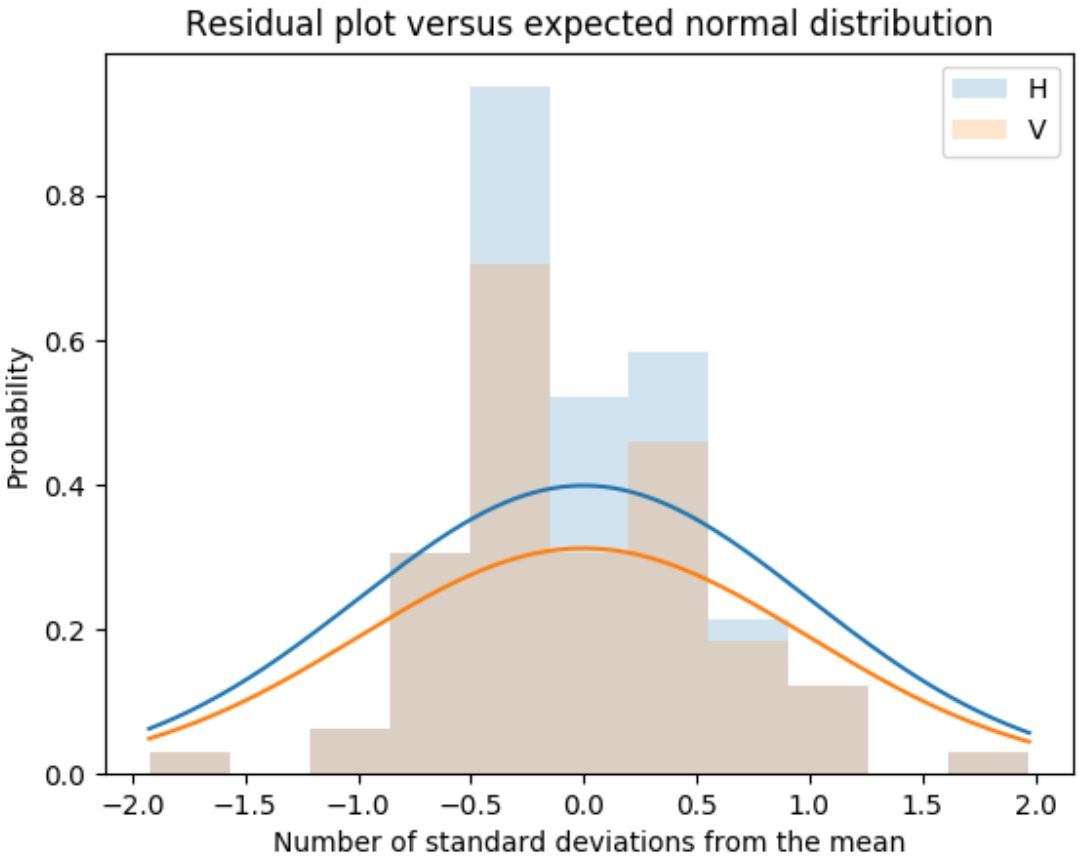
The first example deals with simple PVT fitting. The second example illustrates how powerful it can be to provide non-PVT constraints to the same fitting problem.

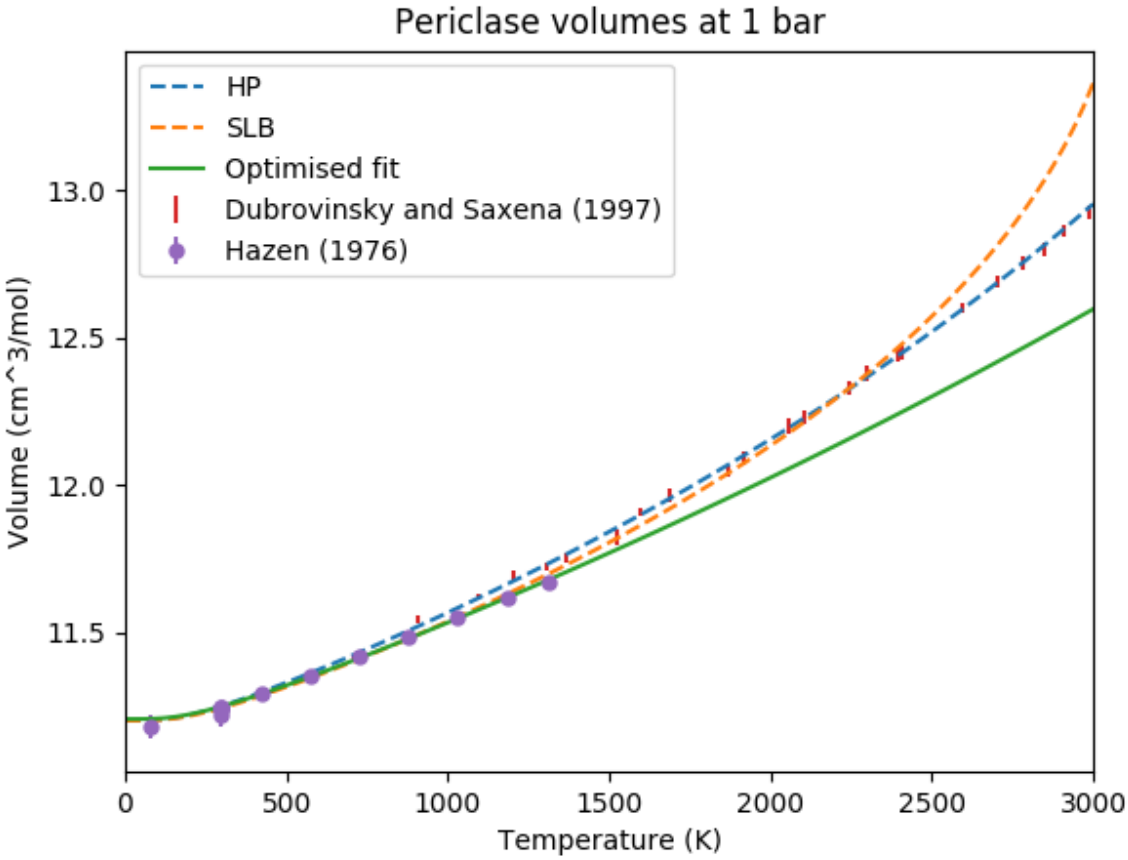
teaches: - least squares fitting

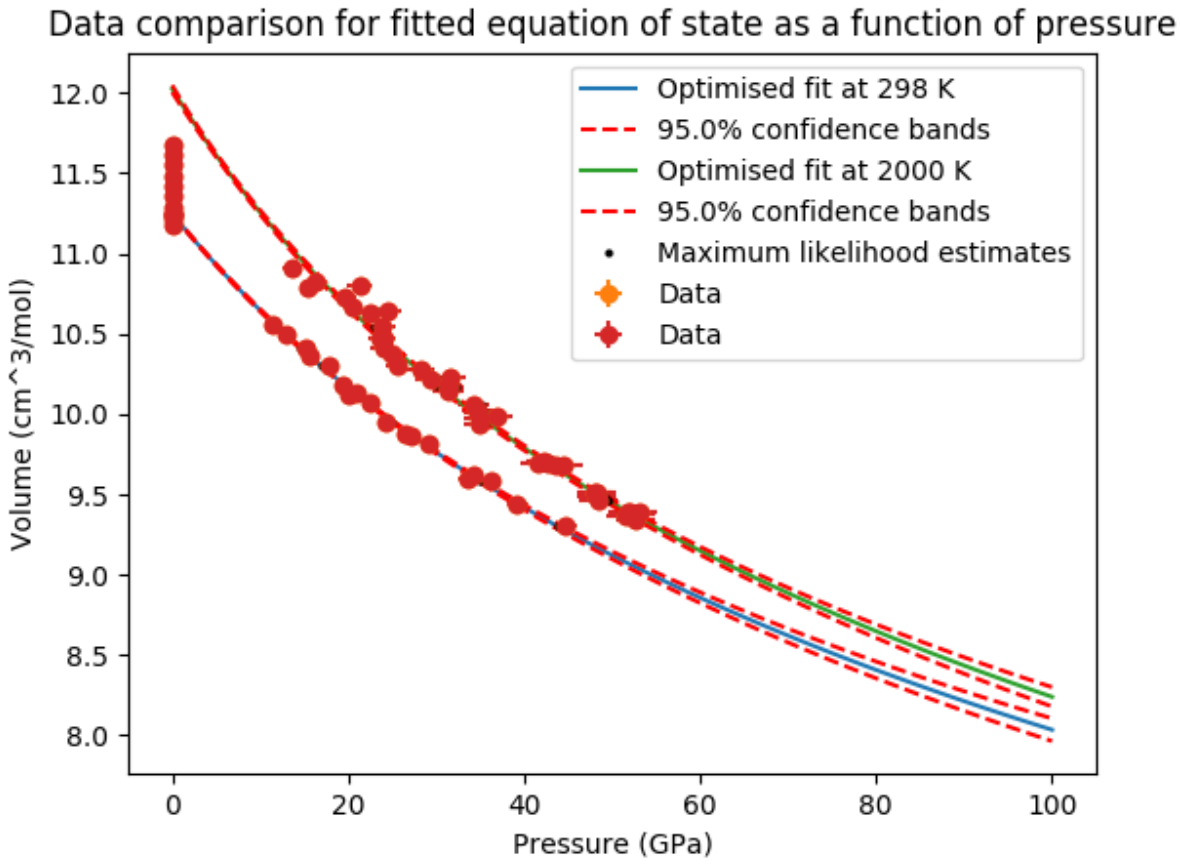
Last seven resulting figures:

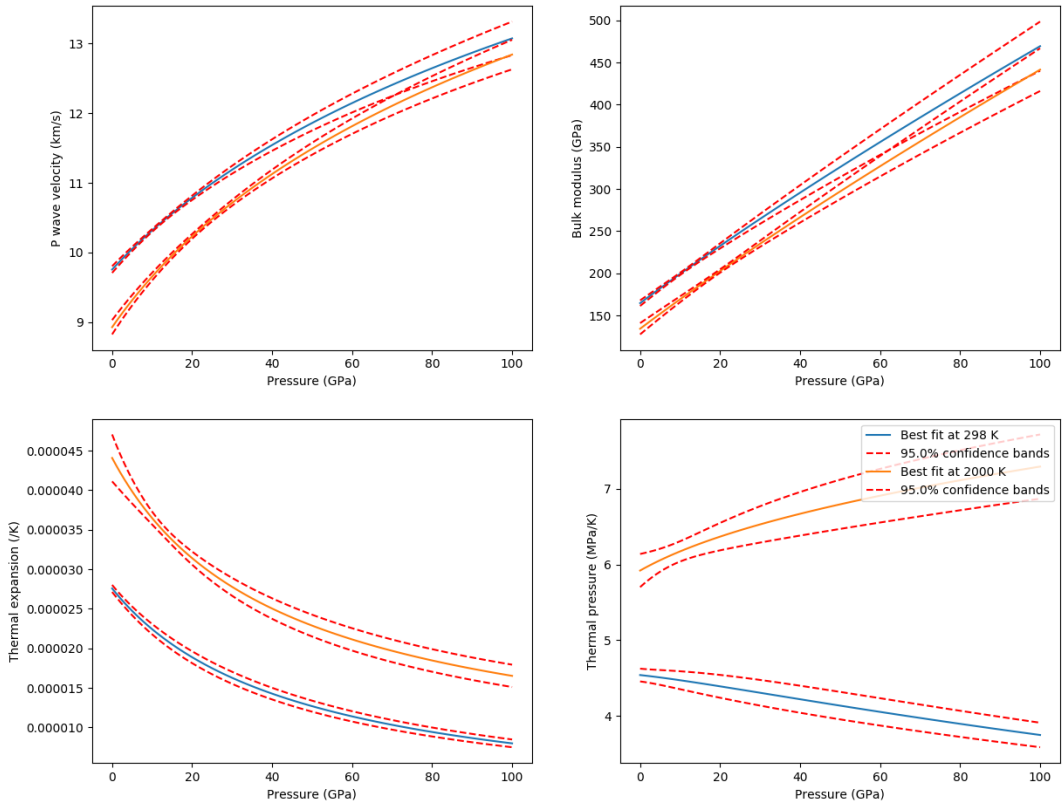

K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)
F_0 (x 10^3)
V_0 (x 10^-5)+1.124
K_0 (x 10^11)
Kprime_0 (x 10^0)
grueneisen_0 (x 10^0)
q_0 (x 10^-1)
Debye_0 (x 10^2)

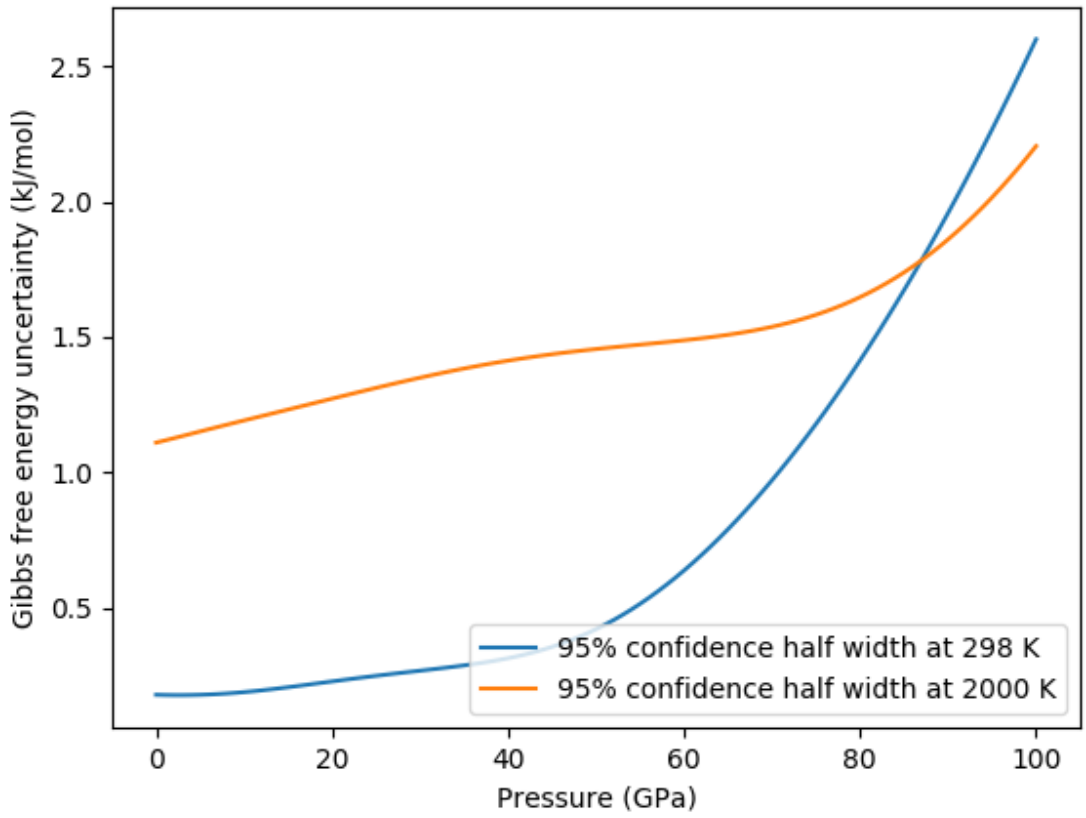

Weighted residual plot for volumes
Temperature (K)
Pressure (GPa)
Misfit (standard deviations)
2500
2000
1500
1000
500
0
0
10
20
30
40
50
3
2
1
0
−1
−2
−3


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



19.3.11 example_fit_solution

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

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

teaches: - least squares fitting for solution data

Resulting figures:

19.3.12 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

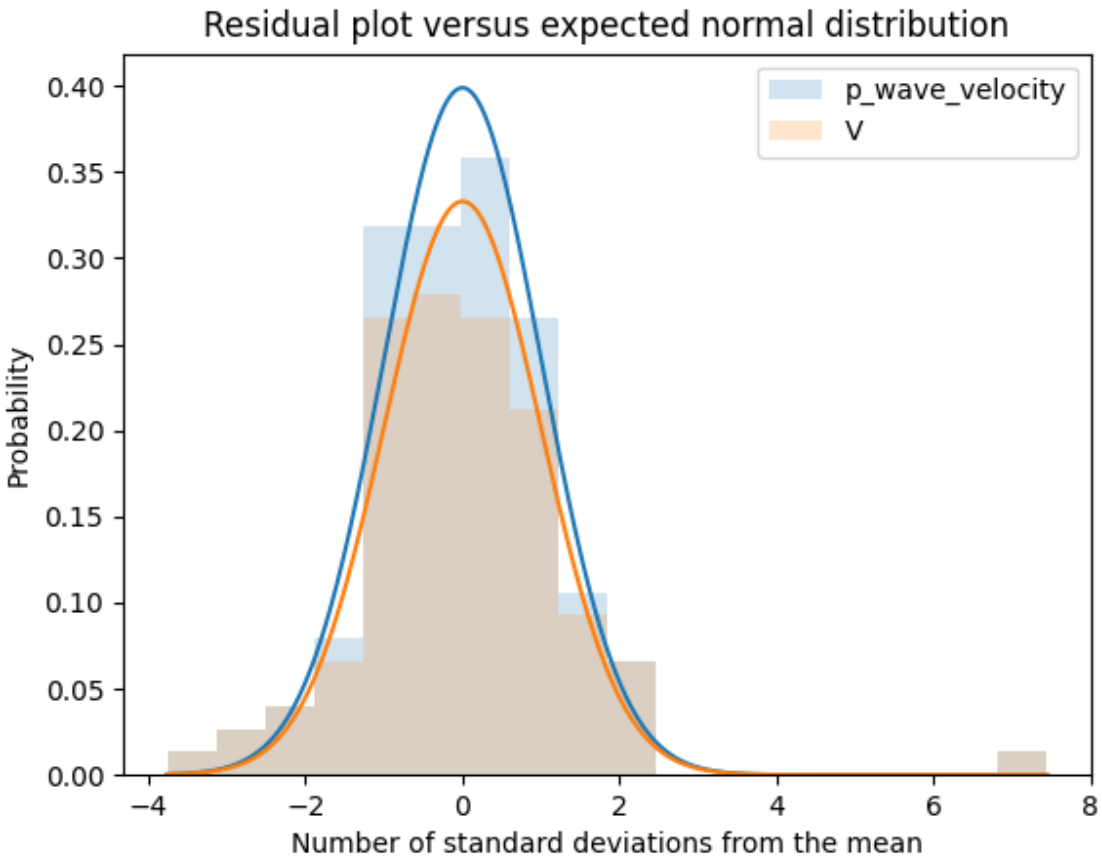


Fig. 1: Data residuals relative to the model fitted using all the provided data.

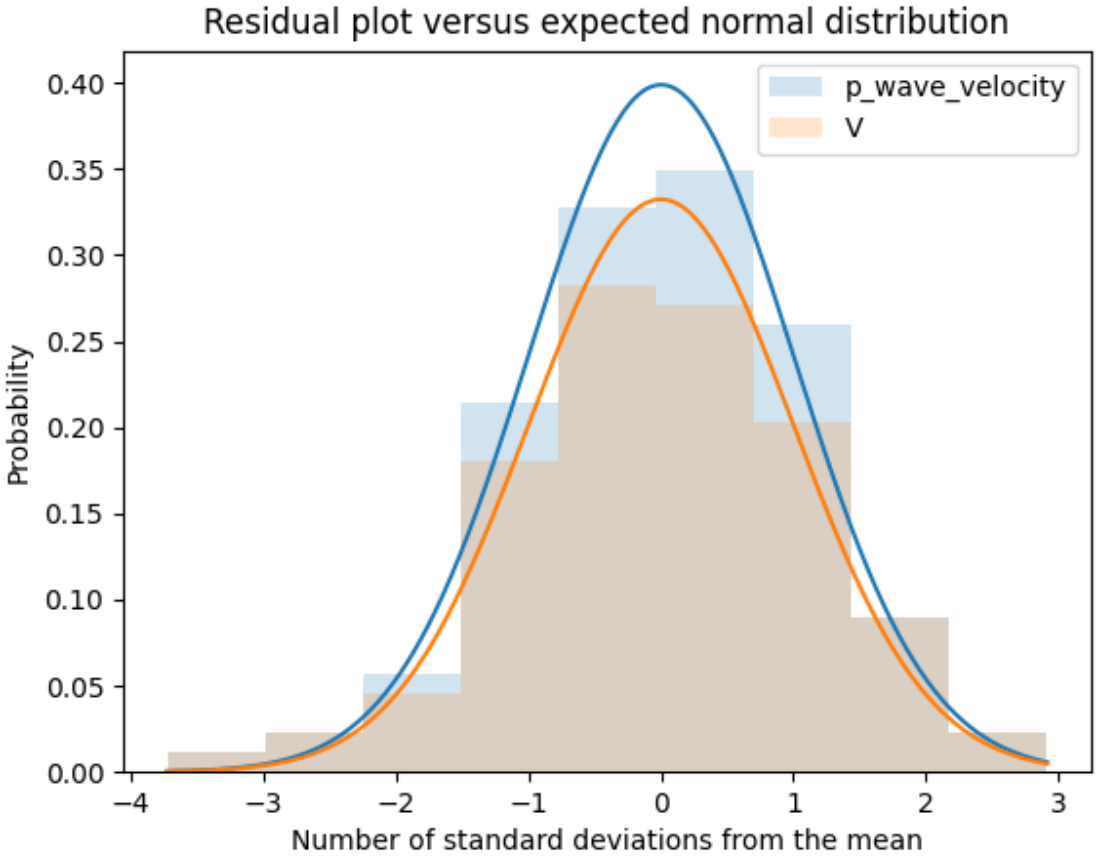


Fig. 2: Data residuals relative to the model fitted using the provided data after semi-automatic removal of spurious data.

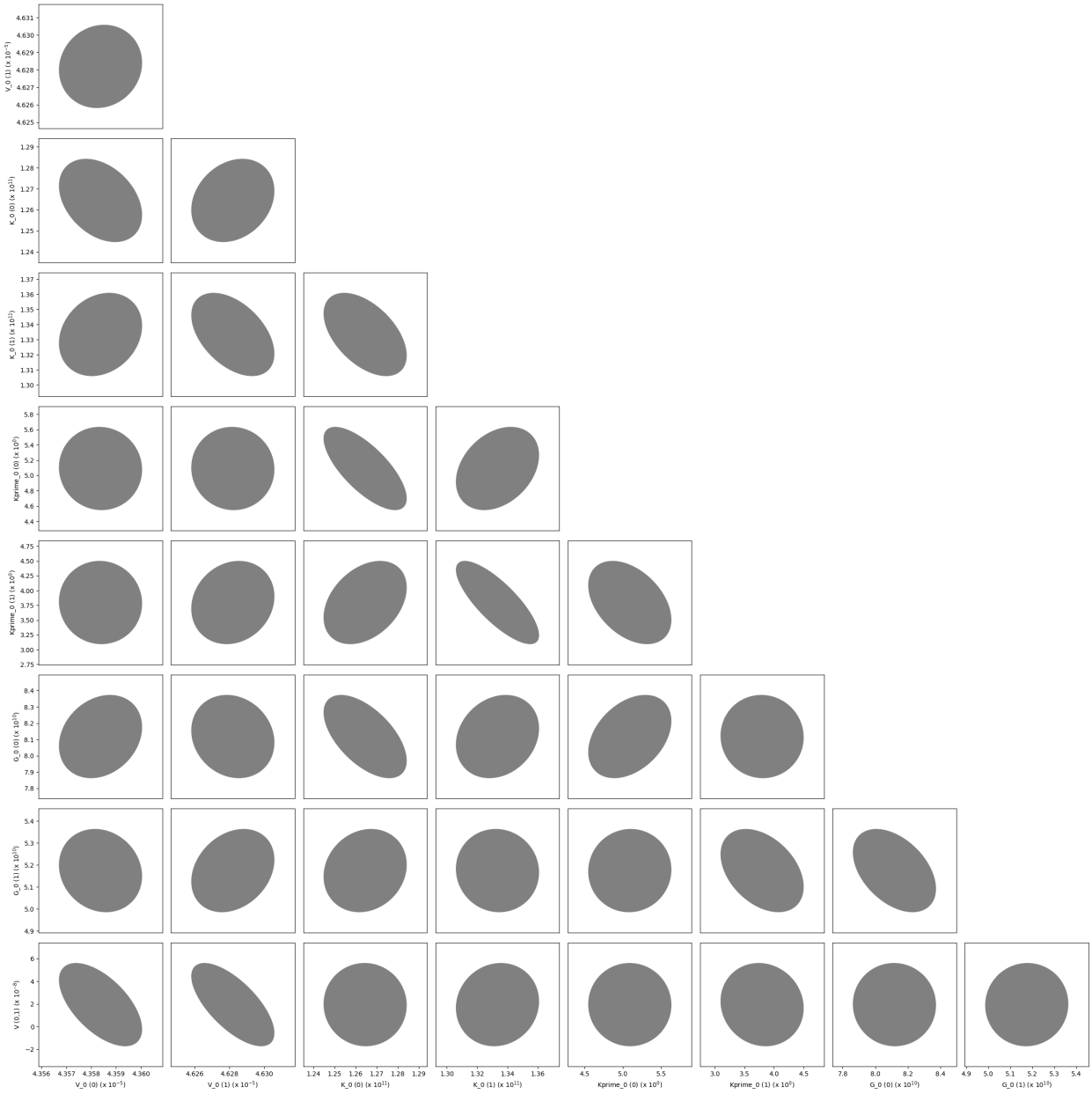


Fig. 3: The variance-covariance matrix of the optimized parameters shown as a corner plot.

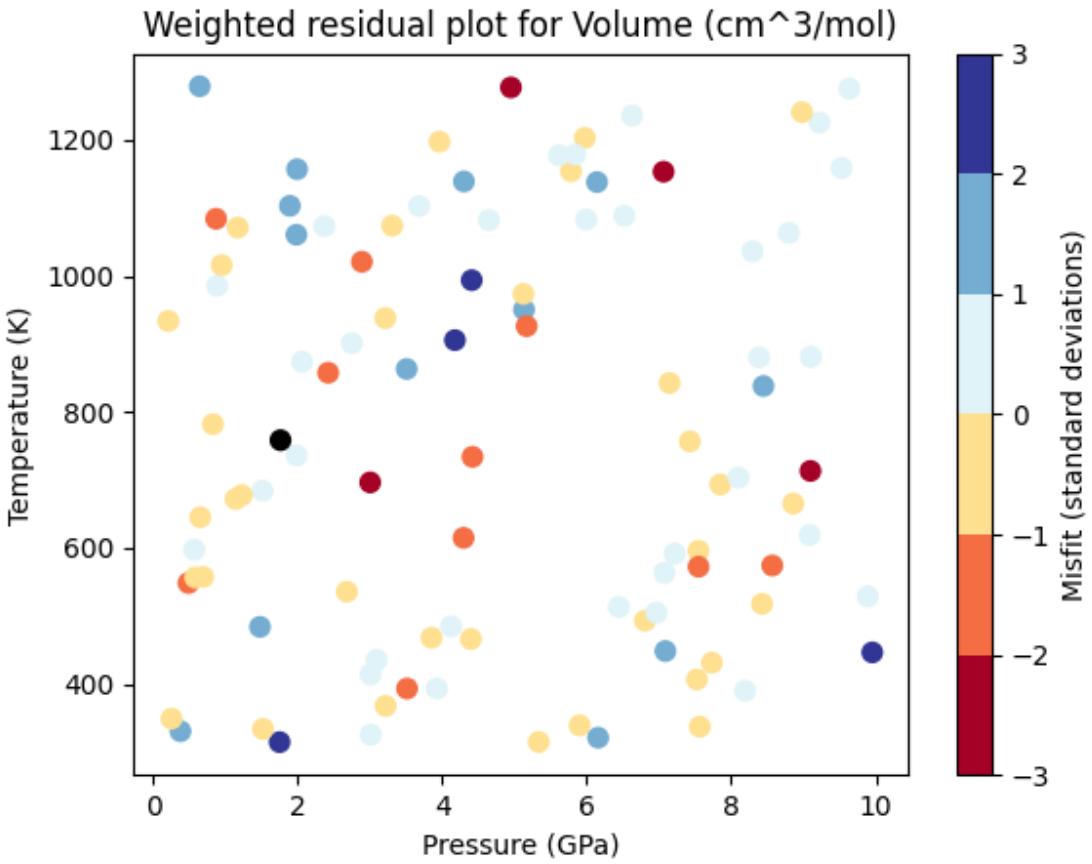


Fig. 4: A P-T plot showing the weighted residuals of each piece of volume data.

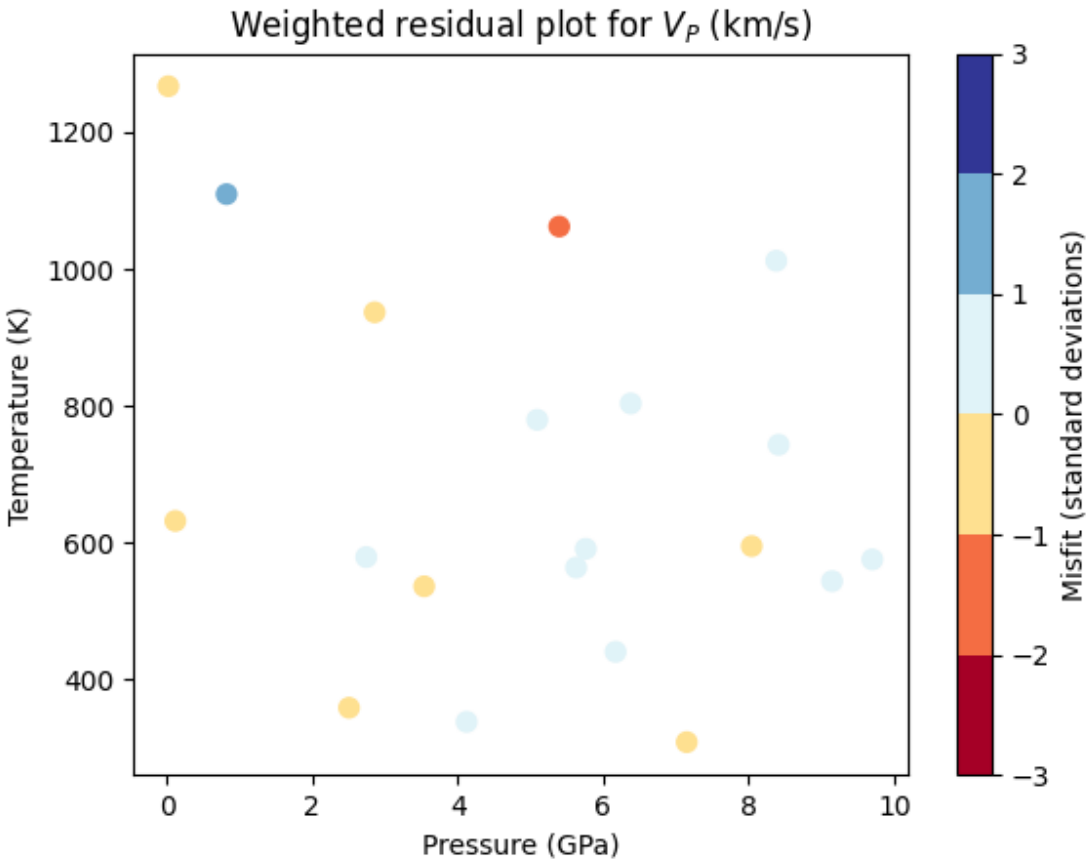


Fig. 5: A P-T plot showing the weighted residuals of each piece of P-wave velocity data.

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 Datasets*
- `burnman.Composite`
- `burnman.equilibrate()`

Resulting figures:

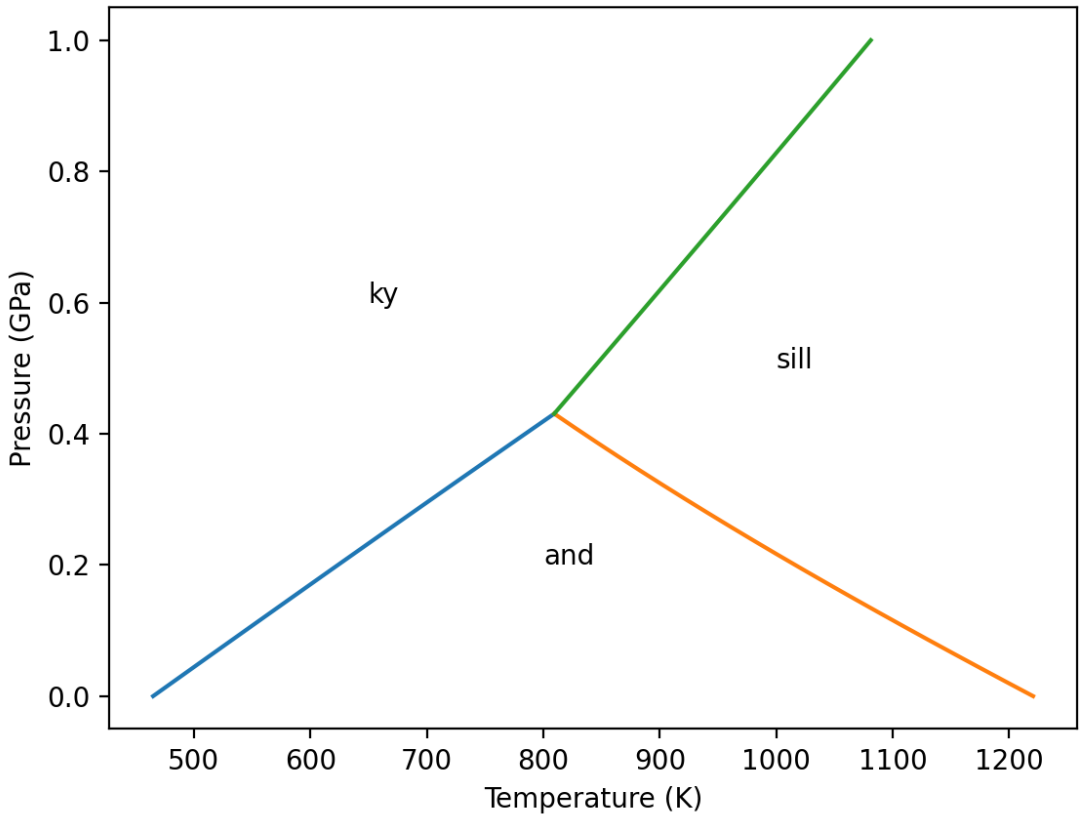


Fig. 6: The classic aluminosilicate diagram.

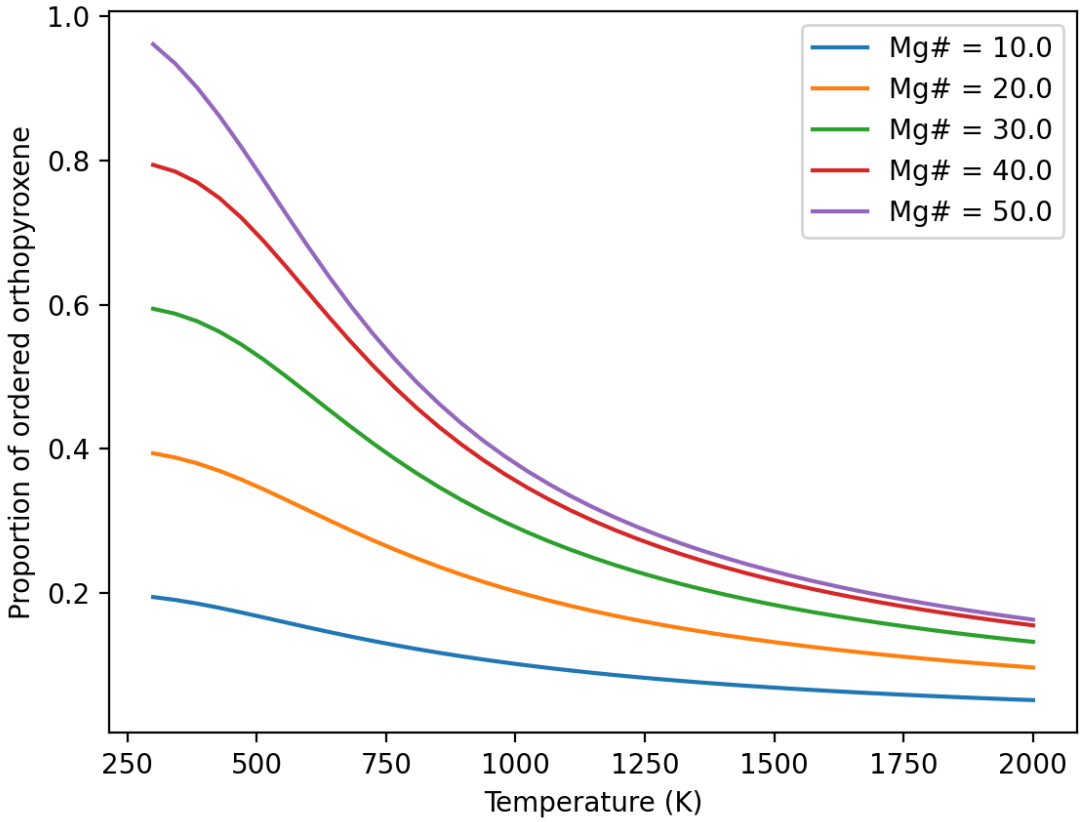


Fig. 7: Ordering in two site orthopyroxene.

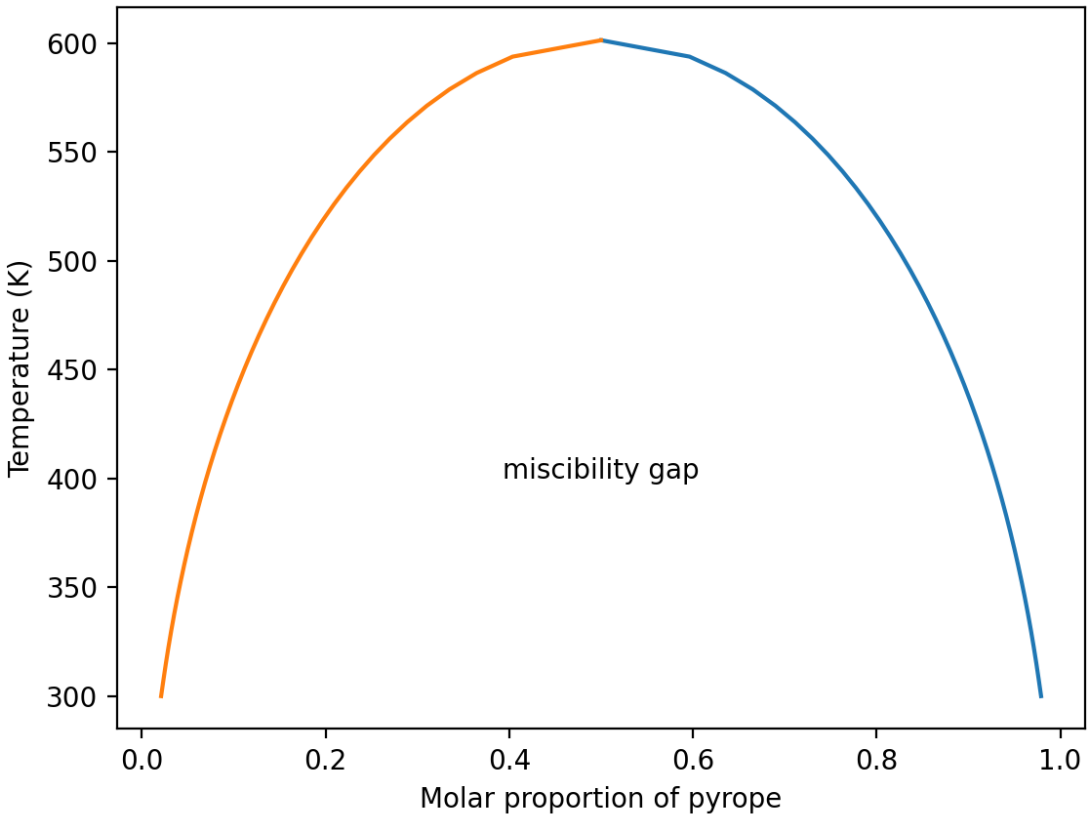


Fig. 8: Miscibility in the pyrope-grossular garnet system

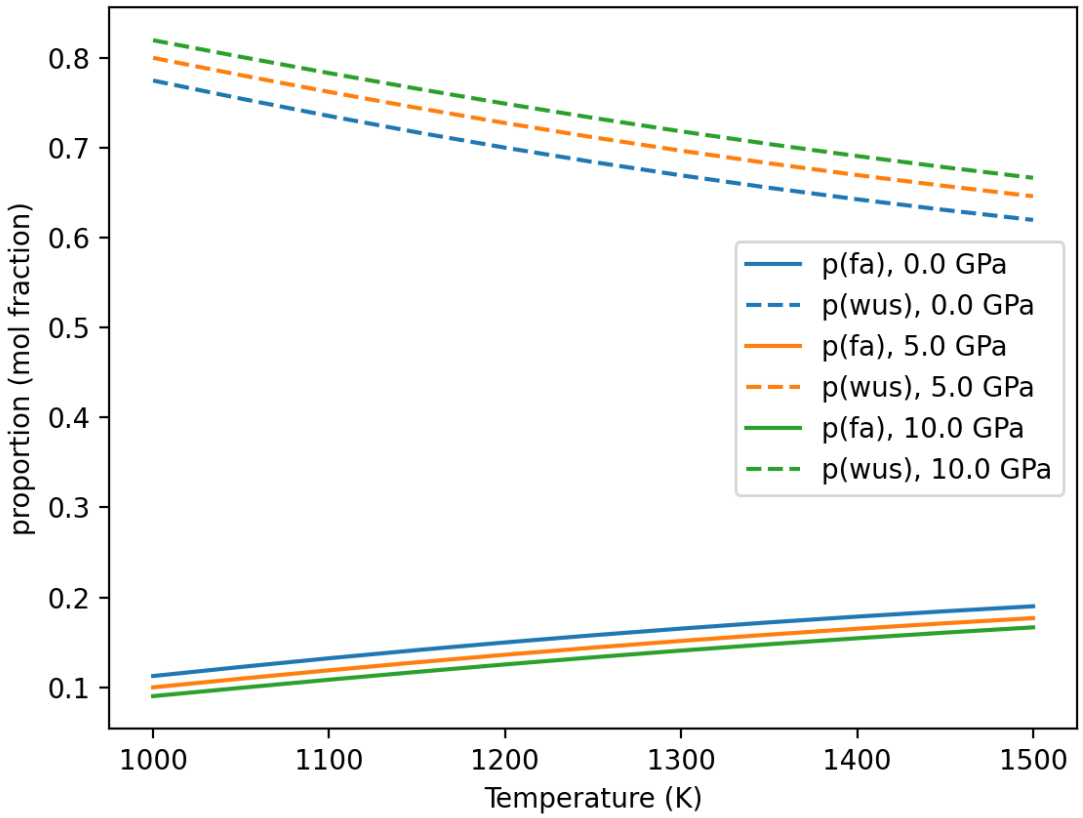


Fig. 9: Mg-Fe partitioning between olivine and ferropericlase.

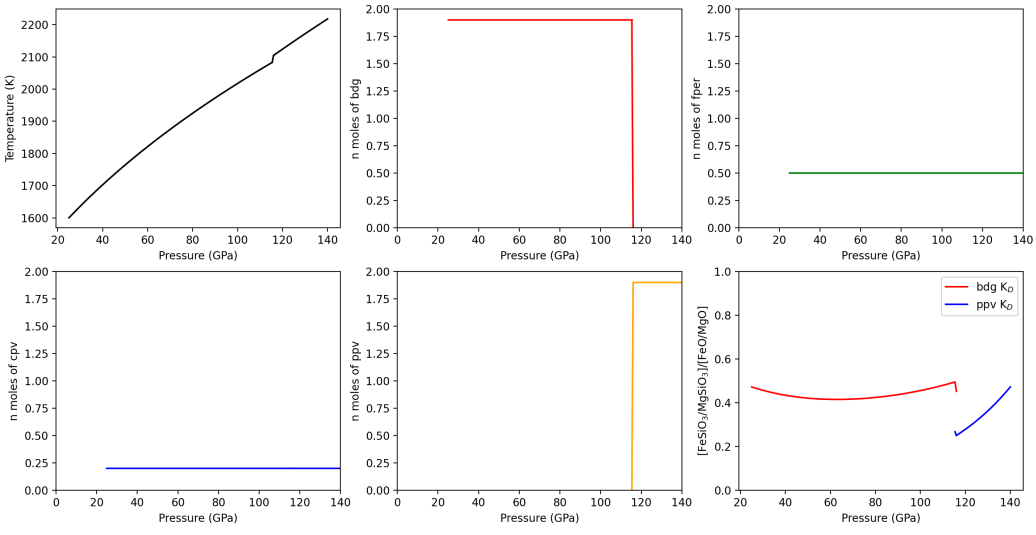


Fig. 10: Phase equilibria in the lower mantle.

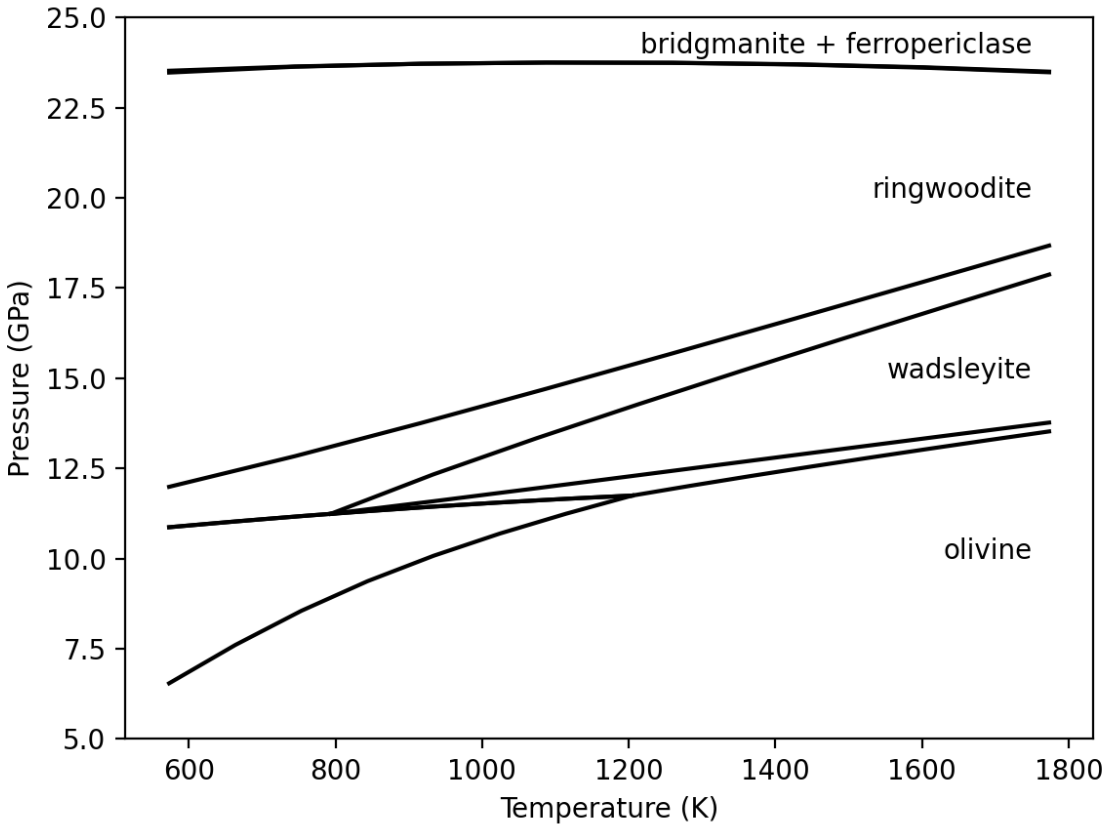


Fig. 11: A P-T pseudosection for a composition of Fe0.2Mg1.8SiO4 (fo90).

19.3.13 example_olivine_binary

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

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

Uses:

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

Resulting figures:

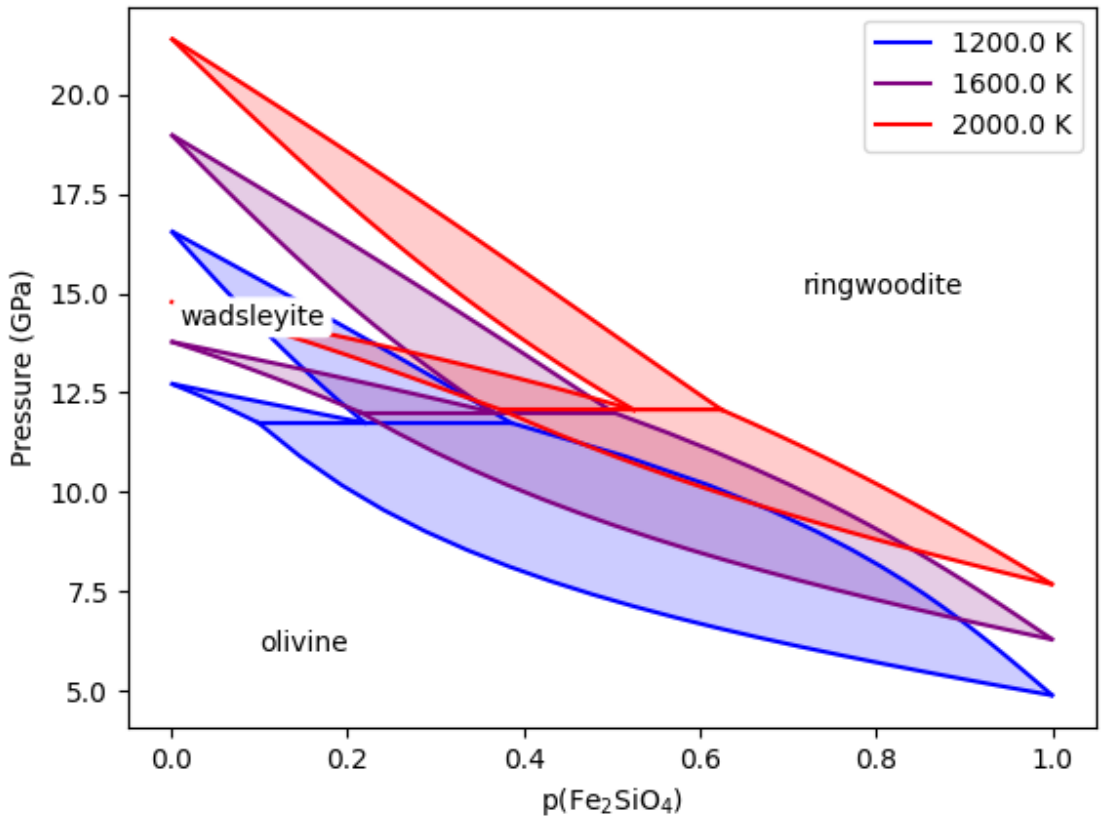


Fig. 12: The olivine polymorph binary phase diagram using the thermodynamic models of Stixrude and Lithgow-Bertelloni (2011).

19.4 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]

19.4.1 paper_averaging

This script reproduces [CHRU14], Figure 2.

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

See [WDOConnell76] for explanations of each averaging scheme.

requires: - geotherms - compute seismic velocities

teaches: - averaging

19.4.2 paper_benchmark

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

19.4.3 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*)

19.4.4 paper_incorrect_averaging

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

19.4.5 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

19.4.6 paper_onefit

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

19.4.7 paper_uncertain

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

19.5 Misc or work in progress

19.5.1 example_grid

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

19.5.2 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

TWENTY

AUTOGENERATED FULL API

20.1 Materials

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

20.1.1 Material Base Class

class burnman.**Material**

Bases: object

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

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

property name

Human-readable name of this material.

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

set_method(*method*)

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

> **Note**
>
> Needs to be implemented in derived classes.

to_string()

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

Returns

A human-readable name of the material.

Return type

str

debug_print(*indent=''*)

Print a human-readable representation of this Material.

print_minerals_of_current_state()

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

set_state(*pressure*, *temperature*)

Set the material to the given pressure and temperature.

Parameters

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

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

reset()

Resets all cached material properties.

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

copy()

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

Returns

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

Return type

tuple

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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

- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property pressure

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

> **Note**
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

property temperature

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

> **Note**
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

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

Returns
The internal energy in [J/mol].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

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

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

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

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_mass

Returns molar mass of the mineral.

> **Note**
>
> Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

`property mass`

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

`property molar_volume`

Returns molar volume of the mineral.

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

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

Return type
float

`property density`

Returns the density of this material.

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

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

Return type
float

`property molar_entropy`

Returns molar entropy of the mineral.

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

Returns
Entropy in [J/K/mol].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the mineral.

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

Returns
Enthalpy in [J/mol].

Return type
float

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the material.

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

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property isentropic_bulk_modulus_reuss

Returns the adiabatic bulk modulus of the mineral.

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

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property isothermal_compressibility_reuss

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

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

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_reuss

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

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

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property shear_modulus

Returns shear modulus of the mineral.

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

Returns
Shear modulus in [Pa].

Return type
float

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

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

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

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

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the mineral.

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

Returns
Shear wave speed in [m/s].

Return type
float

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

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

Returns
Grueneisen parameter [unitless].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the mineral.

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

Returns
Thermal expansivity in [1/K].

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

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

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

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

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

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

Return type
float

property isentropic_thermal_gradient

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property internal_energy

Extensive internal energy of the material (J).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property gibbs

Extensive Gibbs free energy of the material (J).

property enthalpy

Extensive enthalpy of the material (J).

property volume

Extensive volume of the material (m^3).

property entropy

Extensive entropy of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property rho

Alias for *density()*

property S

Alias for *entropy()*

property H

Alias for *enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property G

Alias for *shear_modulus()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property G_eff

Alias for *effective_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 *heat_capacity_v()*

property C_p

Alias for *heat_capacity_p()*

20.2 Perple_X Class

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

Bases: *Material*

PerpleX material class. This class creates a material based on thermodynamic and thermoelastic data read from a 2D PerpleX tab file. This file should be in the standard format output by the werami program, 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 can only be queried after setting the pressure and temperature using set_state(). These properties 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.

Parameters

- **tab_file** (*str*) – Path to the 2D PerpleX tab file.
- **name** (*str*) – Name of the material.

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

(copied from molar_volume):

Returns molar volume of the mineral.

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

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

Return type
float

property molar_enthalpy

(copied from molar_enthalpy):

Returns molar enthalpy of the mineral.

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

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

(copied from molar_entropy):

Returns molar entropy of the mineral.

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

Returns
Entropy in [J/K/mol].

Return type
float

property isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the material.

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

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property isentropic_bulk_modulus_reuss

(copied from isentropic_bulk_modulus_reuss):

Returns the adiabatic bulk modulus of the mineral.

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

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property molar_heat_capacity_p

(copied from molar_heat_capacity_p):

Returns molar heat capacity at constant pressure of the mineral.

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

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

Return type
float

property thermal_expansivity

(copied from thermal_expansivity):

Returns thermal expansion coefficient of the mineral.

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

Returns
Thermal expansivity in [1/K].

Return type
float

property shear_modulus

(copied from shear_modulus):

Returns shear modulus of the mineral.

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

Returns
Shear modulus in [Pa].

Return type
float

property p_wave_velocity

(copied from p_wave_velocity):

Returns P wave speed of the mineral.

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

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

(copied from bulk_sound_velocity):

Returns bulk sound speed of the mineral.

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

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

(copied from shear_wave_velocity):

Returns shear wave speed of the mineral.

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

Returns
Shear wave speed in [m/s].

Return type
float

property molar_gibbs

(copied from molar_gibbs):

Returns the molar Gibbs free energy of the mineral.

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

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_mass

(copied from molar_mass):

Returns molar mass of the mineral.

> **Note**
>
> Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

`property density`

(copied from density):

Returns the density of this material.

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

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

Return type
float

`property molar_internal_energy`

(copied from molar_internal_energy):

Returns the molar internal energy of the mineral.

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

Returns
The internal energy in [J/mol].

Return type
float

`property molar_helmholtz`

(copied from molar_helmholtz):

Returns the molar Helmholtz free energy of the mineral.

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

Returns
Helmholtz free energy in [J/mol].

Return type
float

property isothermal_compressibility_reuss

(copied from isothermal_compressibility_reuss):

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

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

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_reuss

(copied from isentropic_compressibility_reuss):

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

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

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property molar_heat_capacity_v

(copied from molar_heat_capacity_v):

Returns molar heat capacity at constant volume of the mineral.

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

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

Return type
float

property grueneisen_parameter

(copied from grueneisen_parameter):

Returns the grueneisen parameter of the mineral.

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

Returns
Grueneisen parameter [unitless].

Return type
float

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns

Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or `numpy.array`, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

set_method(*method*)

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

> **Note**
>
> Needs to be implemented in derived classes.

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably

bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

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

Returns
A human-readable name of the material.

Return type
str

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

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

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

20.3 Minerals

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

Bases: *Material*

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

This class is available as `burnman.Mineral`.

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

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

property name

Human-readable name of this material.

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

set_method(*equation_of_state*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

to_string()

Returns the name of the mineral class

debug_print(*indent=''*)

Print a human-readable representation of this Material.

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging

schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns

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

Return type

tuple

`set_state()`

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

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

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

`property molar_gibbs`

(copied from molar_gibbs):

Returns the molar Gibbs free energy of the mineral.

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

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_volume

(copied from molar_volume):

Returns molar volume of the mineral.

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

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

Return type
float

property molar_entropy

(copied from molar_entropy):

Returns molar entropy of the mineral.

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

Returns
Entropy in [J/K/mol].

Return type
float

property isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the material.

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

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property molar_heat_capacity_p

(copied from molar_heat_capacity_p):

Returns molar heat capacity at constant pressure of the mineral.

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

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

Return type
float

property thermal_expansivity

(copied from thermal_expansivity):

Returns thermal expansion coefficient of the mineral.

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

Returns
Thermal expansivity in [1/K].

Return type
float

property shear_modulus

(copied from shear_modulus):

Returns shear modulus of the mineral.

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

Returns
Shear modulus in [Pa].

Return type
float

property formula

Returns the chemical formula of the Mineral class

property molar_mass

(copied from molar_mass):

Returns molar mass of the mineral.

> **Note**
>
> Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property density

(copied from density):

Returns the density of this material.

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

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

Return type
float

property molar_internal_energy

(copied from molar_internal_energy):

Returns the molar internal energy of the mineral.

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

Returns
The internal energy in [J/mol].

Return type
float

property molar_helmholtz

(copied from molar_helmholtz):

Returns the molar Helmholtz free energy of the mineral.

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

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_enthalpy

(copied from molar_enthalpy):

Returns molar enthalpy of the mineral.

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

Returns
Enthalpy in [J/mol].

Return type
float

property isentropic_bulk_modulus_reuss

(copied from isentropic_bulk_modulus_reuss):

Returns the adiabatic bulk modulus of the mineral.

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

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property isothermal_compressibility_reuss

(copied from isothermal_compressibility_reuss):

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

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

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_reuss

(copied from isentropic_compressibility_reuss):

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

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

Returns
Adiabatic compressibility in [1/Pa].

Return type
float

property p_wave_velocity

(copied from p_wave_velocity):

Returns P wave speed of the mineral.

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

Returns
P wave speed in [m/s].

Return type
float

property bulk_sound_velocity

(copied from bulk_sound_velocity):

Returns bulk sound speed of the mineral.

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

Returns
Bulk sound velocity in [m/s].

Return type
float

property shear_wave_velocity

(copied from shear_wave_velocity):

Returns shear wave speed of the mineral.

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

Returns
Shear wave speed in [m/s].

Return type
float

property grueneisen_parameter

(copied from grueneisen_parameter):

Returns the grueneisen parameter of the mineral.

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

Returns
Grueneisen parameter [unitless].

Return type
float

property C_p
Alias for *heat_capacity_p()*

property C_v
Alias for *heat_capacity_v()*

property G
Alias for *shear_modulus()*

property G_eff
Alias for *effective_shear_modulus()*

property H
Alias for *enthalpy()*

property K_S
Alias for *isentropic_bulk_modulus_reuss()*

property K_T
Alias for *isothermal_bulk_modulus_reuss()*

property K_eff
Alias for *effective_isentropic_bulk_modulus()*

property P
Alias for *pressure()*

property S
Alias for *entropy()*

property T
Alias for *temperature()*

property V
Alias for *volume()*

property alpha
Alias for *thermal_expansivity()*

property beta_S
Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns

Effective shear modulus in [Pa].

Return type

float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_heat_capacity_v

(copied from molar_heat_capacity_v):

Returns molar heat capacity at constant volume of the mineral.

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

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

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

> **Note**
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

class burnman.**CombinedMineral**(*mineral_list*, *molar_amounts*, *energy_adjustment=[]*, *name='User-created endmember'*)

Bases: *Mineral*

This is the base class for endmembers constructed from a linear combination of other minerals.

Instances of this class should be initialised with a list of Mineral instances, a second list containing the number of moles of each mineral, and (optionally) a third list containing three floats describing a free energy adjustment which is linear in pressure and temperature (i.e. a constant energy [J/mol], entropy [J/K/mol] and volume adjustment [J/Pa/mol or m^3/mol]).

For example, a crude approximation to a bridgmanite model might be bdg = CombinedMineral([per, stv], [1.0, 1.0], [-15.e3, 0., 0.])

This class is available as *burnman.CombinedMineral*.

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 molar_gibbs

Returns Gibbs free energy of the mineral [J] Aliased with self.gibbs

property molar_volume

Returns molar volume of the mineral [m^3/mol] Aliased with self.V

property molar_entropy

Returns entropy of the mineral [J] Aliased with self.S

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the mineral [Pa] Aliased with self.K_T

property shear_modulus

Returns shear modulus of the mineral [Pa] Aliased with self.G

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the mineral [1/K] Aliased with self.alpha

property molar_heat_capacity_p

Returns heat capacity at constant pressure of the mineral [J/K/mol] Aliased with self.C_p

property molar_mass

Returns molar mass of the mineral [kg/mol]

property formula

Returns molar chemical formula of the mineral

property density

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

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol] Aliased with self.energy

property molar_helmholtz

Returns molar Helmholtz free energy of the mineral [J/mol] Aliased with self.helmholtz

property molar_enthalpy

Returns molar enthalpy of the mineral [J/mol] Aliased with self.H

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the mineral [Pa] Aliased with self.K_S

property isothermal_compressibility_reuss

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

property isentropic_compressibility_reuss

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

property p_wave_velocity

Returns P wave speed of the mineral [m/s] Aliased with self.v_p

property bulk_sound_velocity

Returns bulk sound speed of the mineral [m/s] Aliased with self.v_phi

property shear_wave_velocity

Returns shear wave speed of the mineral [m/s] Aliased with self.v_s

property grueneisen_parameter

Returns grueneisen parameter of the mineral [unitless] Aliased with self.gr

property molar_heat_capacity_v

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

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns

Effective shear modulus in [Pa].

Return type

float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property name

Human-readable name of this material.

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

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

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

reset()
Resets all cached material properties.

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

property rho
Alias for *density()*

set_method(*equation_of_state*)
Set the equation of state to be used for this mineral. Takes a string corresponding to any of the pre-defined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)
This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

`to_string()`
Returns the name of the mineral class

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

> **Note**
>
> Needs to be implemented in derived classes.

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

Return type
tuple

`property v_p`
Alias for *p_wave_velocity()*

`property v_phi`
Alias for *bulk_sound_velocity()*

`property v_s`
Alias for *shear_wave_velocity()*

`property volume`
Extensive volume of the material (m^3).

20.4 Solutions

`class burnman.Solution(`*name=None*, *solution_model=None*, *molar_fractions=None*`)`

Bases: *Mineral*

The Solution class has overall responsibility for calculating thermodynamically self-consistent properties of a solution phase.

In this class, solutions are treated as a collection of independent endmember phases (components), plus a model that describes the excess properties of the solution. It stores the current state (pressure, temperature and molar fractions of all the endmembers) of the solution, and interrogates a SolutionModel object to calculate the excess properties of the solution. It then combines these with the properties of the individual endmember phases (also stored in the SolutionModel) to calculate the overall properties of the solution. This is done using standard thermodynamic relations:

$$G = \sum_i X_i G_i + G^{\text{excess}}$$
$$V = \sum_i X_i V_i + V^{\text{excess}}$$
$$S = \sum_i X_i S_i + S^{\text{excess}}$$
$$C_p = \sum_i X_i C_{p,i} + C_p^{\text{excess}}$$
$$\alpha V = \sum_i X_i \alpha_i V_i + (\alpha V)^{\text{excess}}$$
$$\frac{V}{K_T} = \sum_i X_i \frac{V_i}{K_{T,i}} + \left(\frac{V}{K_T}\right)^{\text{excess}}$$

After instantiation, the user can set the composition of the solution using the set_composition() method, and the pressure and temperature using the set_state() method.

Instantiation of a Solution object can be achieved in just a few lines:

```
from burnman import Solution, SolutionModel

# Create a SolutionModel object (explained in the documentation for that␣
↪class)
solution_model = SolutionModel(...)

# Create a Solution object
solution = Solution(name="MySolution", solution_model=solution_model, molar_
↪fractions=...)
```

where the arguments are as follows:

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.SolutionModel*) – The SolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The initial molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

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

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters
molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

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

property formula

Returns molar chemical formula of the solution. :rtype: Counter

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies
- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property activities
Returns a list of endmember activities [unitless].

property activity_coefficients
Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy
Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

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

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

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

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

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

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

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

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

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

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

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

property molar_helmholtz

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

property molar_mass

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

property excess_volume

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

property molar_volume

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

property density

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

property excess_entropy

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

property molar_entropy

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

property excess_enthalpy

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

property molar_enthalpy

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

property isothermal_bulk_modulus_reuss

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

property isentropic_bulk_modulus_reuss

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

property isothermal_compressibility_reuss

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

property isentropic_compressibility_reuss

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

property shear_modulus

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

property p_wave_velocity

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

property bulk_sound_velocity

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

property shear_wave_velocity

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

property grueneisen_parameter

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

property thermal_expansivity

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

property molar_heat_capacity_v

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

property molar_heat_capacity_p

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

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns

An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type

numpy.ndarray

property n_reactions

The number of reactions in reaction_basis.

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type
numpy.ndarray

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

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector $b = [b_C, b_O]$, the constraint $1 * b_C - 2 * b_O = 0$ must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property endmember_names

A list of names for all the endmembers in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

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

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns

Temperature in [K].

Return type

float

to_string()

Returns the name of the mineral class

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

Returns

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

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

burnman.**SolidSolution**

alias of *Solution*

class burnman.**ElasticSolution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all Elastic solutions. Site occupancies, endmember activities and the constant and volume and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state() and set_composition.

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

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and V derivatives in J/K/mol and Pa/mol.

The parameters are relevant to all Elastic solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.ElasticSolutionModel*) – The ElasticSolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

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

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or

'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

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

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property partial_gibbs

Returns endmember partial molar Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial molar volumes [m^3/mol]. Property specific to solutions.

property partial_entropies

Returns endmember partial molar entropies [J/K/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property molar_helmholtz

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

property molar_gibbs

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

property molar_mass

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

property excess_pressure

Returns excess pressure of the solution [Pa]. Specific property for solutions.

property molar_volume

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

property density

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

property excess_entropy

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

property molar_entropy

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

property excess_enthalpy

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

property molar_enthalpy

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

property isothermal_bulk_modulus_reuss

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

property isentropic_bulk_modulus_reuss

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

property isothermal_compressibility_reuss

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

property isentropic_compressibility_reuss

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

property shear_modulus

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

property p_wave_velocity

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

property bulk_sound_velocity

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

property shear_wave_velocity

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

property grueneisen_parameter

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

property thermal_expansivity

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

property molar_heat_capacity_v

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

property molar_heat_capacity_p

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

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

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

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns

dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type

float

property mass

Returns the mass of this material.

Returns

Mass in [kg].

Return type

float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

> **Note**
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

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

should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns

Temperature in [K].

Return type

float

to_string()

Returns the name of the mineral class

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

Returns

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

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

burnman.**ElasticSolidSolution**

alias of *ElasticSolution*

20.5 Mineral helpers

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

Bases: *Composite*

Helper class that makes a mineral that switches between two materials (for low and high spin) based on some transition pressure [Pa]

debug_print(*indent=''*)

Print a human-readable representation of this Material.

set_state(*pressure*, *temperature*)

Update the material to the given pressure [Pa] and temperature [K].

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property bulk_sound_velocity

Returns bulk sound speed of the composite [m/s] Aliased with self.v_phi

chemical_potential(*components=None*)

Returns the chemical potentials of the currently defined components in the composite. Raises an exception if the assemblage is not equilibrated.

Parameters

components (*list of dictionaries*) – List of formulae of the desired components. If not specified, the method uses the components specified by a previous call to set_components.

Returns

The chemical potentials of the desired components in the equilibrium composite.

Return type

numpy.array of floats

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the composite.

copy()

property density

Compute the density of the composite based on the molar volumes and masses Aliased with self.rho

property dependent_element_indices

The element indices not included in the independent list.

property effective_isentropic_bulk_modulus

Returns the effective isotropic, isentropic bulk modulus of the composite [Pa] Aliased with self.K_eff

property effective_shear_modulus

Returns the effective isotropic shear modulus of the mineral [Pa] Aliased with self.G_eff

property elements

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

property endmember_formulae

A list of the formulae in the composite.

property endmember_names

A list of the endmember names contained in the composite. Mineral names are returned as given in Mineral.name. Solution endmember names are given in the format *Mineral.name in Solution.name*.

property endmember_partial_entropies

Returns the partial entropies for all the endmember minerals in the Composite

property endmember_partial_gibbs

Returns the partial Gibbs energies for all the endmember minerals in the Composite

property endmember_partial_volumes

Returns the partial volumes 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 enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

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*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property formula

Returns extensive chemical formula of the composite

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the composite [unitless]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium.

Aliased with self.gr

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the mineral [Pa]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isentropically compressed, since heat exchange between phases may be required to maintain the same temperature in all phases.

Aliased with self.K_S

property isentropic_compressibility_reuss

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

property isentropic_thermal_gradient

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the composite [Pa] Aliased with self.K_T

property isothermal_compressibility_reuss

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

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

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]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isochorically heated, since differential expansion between phases may be required to maintain the same pressure in all phases.

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 number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

Returns P wave speed of the composite [m/s] Aliased with self.v_p

property phase_masses

Returns total masses of each phase in the composite

property phase_volumes

Returns total volumes of each phase in the composite

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns

Pressure in [Pa].

Return type

float

print_minerals_of_current_state()

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

property reaction_affinities

Returns the affinities corresponding to each reaction in reaction_basis

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property reaction_basis_as_strings

Returns a list of string representations of all the reactions in reaction_basis.

property reduced_stoichiometric_array

The stoichiometric array including only the independent elements

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

set_averaging_scheme(*averaging_scheme*)

Set the averaging scheme for the moduli in the composite. Default is set to VoigtReussHill, when Composite is initialized.

set_components(*components*)

Sets the components and components_array attributes of the composite material. The components attribute is a list of dictionaries containing the chemical formulae of the components. The components_array attribute is a 2D numpy array describing the linear transformation between endmember amounts and component amounts.

The components do not need to be linearly independent, not do they need to form a complete basis set for the composite. However, it must be possible to obtain the composition of each component from a linear sum of the endmember compositions of the composite. For example, if the composite was composed of MgSiO3 and Mg2SiO4, SiO2 would be a valid component, but Si would not. The method raises an exception if any of the chemical potentials are not defined by the assemblage.

Parameters

components (*list of dictionaries*) – List of formulae of the components.

set_fractions(*fractions*, *fraction_type='molar'*)

Change the fractions of the phases of this Composite. Resets cached properties

Parameters

- **fractions** – list or numpy array of floats molar or mass fraction for each phase.
- **fraction_type** – 'molar' or 'mass' specify whether molar or mass fractions are specified.

set_method(*method*)

set the same equation of state method for all the phases in the composite

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral.

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

Returns
Shear modulus in [Pa].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the composite [m/s] Aliased with self.v_s

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_matrix

An sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient of the composite [1/K] Aliased with self.alpha

to_string()

return the name of the composite

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

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

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

20.6 Anisotropic materials

class burnman.**AnisotropicMaterial**(*rho*, *cijs*)

Bases: *Material*

A base class for anisotropic elastic materials. The base class is initialised with a density and a full isentropic stiffness tensor in Voigt notation. It can then be interrogated to find the values of different properties, such as bounds on seismic velocities. There are also several functions which can be called to calculate properties along directions oriented with respect to the isentropic elastic tensor.

See [MHS11] and https://docs.materialsproject.org/methodology/elasticity/ for mathematical descriptions of each function.

property isentropic_stiffness_tensor

property full_isentropic_stiffness_tensor

property isentropic_compliance_tensor

property full_isentropic_compliance_tensor

property density

Returns the density of this material.

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

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

Return type
float

`property isentropic_bulk_modulus_voigt`

Returns
The Voigt bound on the isentropic bulk modulus [Pa].

Return type
float

`property isentropic_bulk_modulus_reuss`

Returns
The Reuss bound on the isentropic bulk modulus [Pa].

Return type
float

`property isentropic_bulk_modulus_vrh`

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

`property isentropic_shear_modulus_voigt`

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

`property isentropic_shear_modulus_reuss`

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

`property isentropic_shear_modulus_vrh`

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the lateral response to loading.

Return type
float

christoffel_tensor(*propagation_direction*)

Calculates the adiabatic Christoffel tensor.

The isentropic stiffness tensors provided in this and other anisotropic classes correspond to the isentropic derivative of stress with respect to symmetric infinitesimal strain. This corresponds to the isentropic elastic tensor B of Wallace (1967; doi:10.1103/PhysRev.162.776).

The Christoffel (acoustic) matrix is the contraction T_ik = S_ijkl n_j n_l, where S is Wallace's isentropic equation-of-motion tensor (not to be confused with the isentropic compliance tensor). For a unit direction n, contracting S gives T = sym(T_B) + ((n . sigma . n) I - sigma) / 2, where T_B_ik = B_ijkl n_j n_l and sigma is the tension-positive Cauchy stress. This derivation follows from Wallace (1967), using his Equations (2.24), (2.28), and (2.36).

Parameters
propagation_direction – Direction or array of directions with shape (..., 3), expressed in the same frame as the stress and stiffness tensors. Each direction is normalized before contraction.

Returns
Christoffel tensor in Pa, with shape (..., 3, 3).

Return type
numpy.ndarray

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction relative to the stiffness tensor [1/Pa].

Return type
float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction relative to the stiffness tensor [Pa].

Return type
float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor [Pa].

Return type
float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response directions relative to the stiffness tensor [unitless].

Return type
float

wave_velocities(*propagation_direction*)

Returns
The compressional wave velocity, and two shear wave velocities in a given propagation direction [m/s].

Return type
list, containing the wave speeds and directions of particle motion relative to the stiffness tensor

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property bulk_sound_velocity

Returns bulk sound speed of the mineral.

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

Returns

Bulk sound velocity in [m/s].

Return type

float

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns the grueneisen parameter of the mineral.

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

Returns
Grueneisen parameter [unitless].

Return type
float

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_compressibility_reuss

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

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

Returns

Adiabatic compressibility in [1/Pa].

Return type

float

property isentropic_thermal_gradient

Returns

dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type

float

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the material.

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

Returns

Isothermal bulk modulus in [Pa].

Return type

float

property isothermal_compressibility_reuss

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

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

Returns
Isothermal compressibility in [1/Pa].

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the mineral.

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

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

Returns molar entropy of the mineral.

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

Returns
Entropy in [J/K/mol].

Return type
float

property molar_gibbs

Returns the molar Gibbs free energy of the mineral.

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

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the mineral.

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

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

Return type
float

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the mineral.

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

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

Return type
float

property molar_helmholtz

Returns the molar Helmholtz free energy of the mineral.

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

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_internal_energy

Returns the molar internal energy of the mineral.

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

Returns
The internal energy in [J/mol].

Return type
float

property molar_mass

Returns molar mass of the mineral.

> **Note**
> Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property molar_volume

Returns molar volume of the mineral.

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

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

Return type
float

property name

Human-readable name of this material.

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

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

Returns P wave speed of the mineral.

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

Returns
P wave speed in [m/s].

Return type
float

property pressure

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

> **Note**
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

set_method(*method*)

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

> **Note**
>
> Needs to be implemented in derived classes.

set_state(*pressure*, *temperature*)

Set the material to the given pressure and temperature.

Parameters

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

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral.

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

Returns

Shear modulus in [Pa].

Return type
float

`property shear_wave_velocity`

Returns shear wave speed of the mineral.

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

Returns
Shear wave speed in [m/s].

Return type
float

`property temperature`

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

> **Note**
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

`property thermal_expansivity`

Returns thermal expansion coefficient of the mineral.

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

Returns
Thermal expansivity in [1/K].

Return type
float

`to_string()`

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

Returns
A human-readable name of the material.

Return type
str

unroll()

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

Note

Needs to be implemented in derived classes.

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

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

class burnman.**AnisotropicMineral**(*isotropic_mineral*, *cell_parameters*, *anisotropic_parameters*, *frame_convention=None*, *psi_function=None*, *orthotropic=None*)

Bases: *Mineral*, *AnisotropicMaterial*

A class implementing the anisotropic mineral equation of state described in [Myh22]. This class is derived from both Mineral and AnisotropicMaterial, and inherits most of the methods from these classes.

Instantiation of an AnisotropicMineral takes three required arguments; a reference Mineral (i.e. a standard isotropic mineral which provides volume as a function of pressure and temperature), cell_parameters, which give the lengths of the molar cell vectors and the angles between them (see *cell_parameters_to_vectors()*), and an anisotropic parameters object, which should be either a 4D array of anisotropic parameters or a dictionary of parameters which describe the anisotropic behaviour of the mineral. For a description of the physical meaning of the parameters in the 4D array, please refer to the code or to the original paper.

For non-orthotropic materials, the argument frame_convention should be set to define the orientation of the reference frame relative to the crystallographic axes (see *cell_parameters_to_vectors()*).

If the user chooses to define their parameters as a dictionary, they must also provide a function to the psi_function argument that describes how to compute the tensors Psi, dPsidf and dPsidPth (in Voigt form). The function arguments should be f, Pth and params, in that order. The output variables Psi, dPsidf and dPsidth must be returned in that order in a tuple. The user should also explicitly state whether the material is orthotropic or not by supplying a boolean to the orthotropic argument.

States of the mineral can only be queried after setting the pressure and temperature using set_state().

This class is available as `burnman.AnisotropicMineral`.

All the material parameters are expected to be in plain SI units. This means that the elastic moduli should be in Pascals and NOT Gigapascals. Additionally, the cell parameters should be in m/(mol formula unit) and not in unit cell lengths. To convert unit cell lengths given in Angstrom to molar cell parameters you should multiply by 10^(-10) * (N_a / Z)^1/3, where N_a is Avogadro's number and Z is the number of formula units per unit cell. You can look up Z in many places, including www.mindat.org.

Finally, it is assumed that the unit cell of the anisotropic material is aligned in a particular way relative to the coordinate axes (the anisotropic_parameters are defined relative to the coordinate axes). The crystallographic a-axis is assumed to be parallel to the first spatial coordinate axis, and the crystallographic b-axis is assumed to be perpendicular to the third spatial coordinate axis.

property name

Human-readable name of this material.

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

standard_psi_function(*f*, *Pth*, *params*)

set_state()

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

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

property deformation_gradient_tensor

Returns

The deformation gradient tensor describing the deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature) to its current state. Note that this does not include the rotation of the mineral back into the crystal frame, which is described by the rotation_matrix property.

Return type

numpy.array (2D)

property unrotated_cell_vectors

Returns

The vectors of the cell [m] constructed from one mole of formula units after deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (2D)

property rotation_matrix

Returns

The rotation from the unrotated frame into the crystallographic frame. Column vectors transform as `R @ v` and second-order tensors as `R @ A @ R.T`. For orthotropic minerals, this is the identity matrix.

Return type

numpy.array (2D)

property cell_vectors

Returns

The vectors of the cell constructed from one mole of formula units [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (2D)

property cell_parameters

Returns

The molar cell parameters of the mineral, given in standard form: [a, b, c, α, β, γ], where the first three floats are the lengths of the vectors in [m] defining the cell constructed from one mole of formula units. The last three floats are angles between vectors (given in radians). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (1D)

property shear_modulus

Anisotropic minerals do not (in general) have a single shear modulus. This function returns a NotImplementedError. Users should instead consider directly querying the elements in the isothermal_stiffness_tensor or isentropic_stiffness_tensor.

property K_T

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 isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the material.

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

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property K_S

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 isentropic_bulk_modulus_reuss

(copied from isentropic_bulk_modulus_reuss):

Returns the adiabatic bulk modulus of the mineral.

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

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property beta_T

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider using isothermal_compressibility_tensor, isothermal_compressibility_reuss or isothermal_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isothermal_compliance_tensor.

property beta_S

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider either using isentropic_compressibility_tensor, isentropic_compressibility_reuss or isentropic_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isentropic_compliance_tensor.

property isothermal_bulk_modulus_voigt

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

Return type
float

property isothermal_compressibility_reuss

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

Return type
float

property isothermal_compressibility_voigt

Returns
The Voigt bound on the isothermal compressibility in [1/Pa].

Return type
float

property isentropic_bulk_modulus_voigt

Returns
The Voigt bound on the isentropic bulk modulus in [Pa].

Return type
float

property isentropic_compressibility_reuss

Returns
The Reuss bound on the isentropic compressibility in [1/Pa].

Return type
float

property isentropic_compressibility_voigt

Returns
The Voigt bound on the isentropic compressibility in [1/Pa].

Return type
float

property isothermal_compliance_tensor

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

Return type
numpy.array (2D)

property thermal_expansivity_tensor

Returns
The tensor of thermal expansivities [1/K].

Return type
numpy.array (2D)

property isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

property full_isothermal_compliance_tensor

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

Return type
numpy.array (4D)

property full_isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property full_isentropic_compliance_tensor

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

Return type
numpy.array (4D)

property isentropic_compliance_tensor

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

Return type
numpy.array (2D)

property isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property full_isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property grueneisen_tensor

Returns
The grueneisen tensor [unitless]. This is defined by [BM67] as $\mathbb{C}_{\mathrm{N}ijkl}\alpha_{kl}V/C_P$.

Return type
numpy.array (2D)

property molar_heat_capacity_v

Returns
The isochoric, hydrostatic heat capacity [J/K/mol].

Return type
float

property grueneisen_parameter

Returns
The scalar grueneisen parameter [unitless].

Return type
float

property isothermal_compressibility_tensor

Returns
The isothermal compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property isentropic_compressibility_tensor

Returns
The isentropic compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property bulk_sound_velocity

(copied from bulk_sound_velocity):

Returns bulk sound speed of the mineral.

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

Returns

Bulk sound velocity in [m/s].

Return type

float

christoffel_tensor(*propagation_direction*)

Calculates the adiabatic Christoffel tensor.

The isentropic stiffness tensors provided in this and other anisotropic classes correspond to the isentropic derivative of stress with respect to symmetric infinitesimal strain. This corresponds to the isentropic elastic tensor B of Wallace (1967; doi:10.1103/PhysRev.162.776).

The Christoffel (acoustic) matrix is the contraction T_ik = S_ijkl n_j n_l, where S is Wallace's isentropic equation-of-motion tensor (not to be confused with the isentropic compliance tensor). For a unit direction n, contracting S gives T = sym(T_B) + ((n . sigma . n) I - sigma) / 2, where T_B_ik = B_ijkl n_j n_l and sigma is the tension-positive Cauchy stress. This derivation follows from Wallace (1967), using his Equations (2.24), (2.28), and (2.36).

Parameters

propagation_direction – Direction or array of directions with shape (..., 3), expressed in the same frame as the stress and stiffness tensors. Each direction is normalized before contraction.

Returns

Christoffel tensor in Pa, with shape (..., 3, 3).

Return type

numpy.ndarray

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property density

(copied from density):

Returns the density of this material.

Note

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

Returns

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

Return type

float

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property formula

Returns the chemical formula of the Mineral class

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the lateral response to loading.

Return type
float

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction relative to the stiffness tensor [1/Pa].

Return type
float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response directions relative to the stiffness tensor [unitless].

Return type
float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor [Pa].

Return type
float

property isentropic_shear_modulus_reuss

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_voigt

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction relative to the stiffness tensor [Pa].

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

(copied from molar_enthalpy):

Returns molar enthalpy of the mineral.

Note

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

Returns
Enthalpy in [J/mol].

Return type
float

property molar_entropy

(copied from molar_entropy):

Returns molar entropy of the mineral.

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

Returns
Entropy in [J/K/mol].

Return type
float

property molar_gibbs

(copied from molar_gibbs):

Returns the molar Gibbs free energy of the mineral.

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

Returns
Gibbs free energy in [J/mol].

Return type
float

property molar_heat_capacity_p

(copied from molar_heat_capacity_p):

Returns molar heat capacity at constant pressure of the mineral.

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

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

Return type
float

property molar_helmholtz

(copied from molar_helmholtz):

Returns the molar Helmholtz free energy of the mineral.

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

Returns
Helmholtz free energy in [J/mol].

Return type
float

property molar_internal_energy

(copied from molar_internal_energy):

Returns the molar internal energy of the mineral.

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

Returns
The internal energy in [J/mol].

Return type
float

property molar_mass

(copied from molar_mass):

Returns molar mass of the mineral.

> **Note**
>
> Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
float

property molar_volume

(copied from molar_volume):

Returns molar volume of the mineral.

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

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

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

(copied from p_wave_velocity):

Returns P wave speed of the mineral.

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

Returns
P wave speed in [m/s].

Return type
float

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

set_method(*equation_of_state*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_wave_velocity

(copied from shear_wave_velocity):

Returns shear wave speed of the mineral.

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

Returns
Shear wave speed in [m/s].

Return type
float

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

(copied from thermal_expansivity):

Returns thermal expansion coefficient of the mineral.

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

Returns
Thermal expansivity in [1/K].

Return type
float

property thermal_stress_tensor

Returns
The change in stress with temperature at constant strain [Pa/K].

Return type
numpy.array (2D)

to_string()

Returns the name of the mineral class

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

Returns

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

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

wave_velocities(*propagation_direction*)

Returns

The compressional wave velocity, and two shear wave velocities in a given propagation direction [m/s].

Return type

list, containing the wave speeds and directions of particle motion relative to the stiffness tensor

property molar_isometric_heat_capacity

Returns

The molar heat capacity at constant strain [J/K/mol].

Return type

float

check_standard_parameters(*anisotropic_parameters*)

burnman.**cell_parameters_to_vectors**(*cell_parameters*, *frame_convention*)

Converts cell parameters to unit cell vectors.

Parameters

- **cell_parameters** (*numpy.array (1D)*) – An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.
- **frame_convention** (*list of three integers*) – A list (c) defining the reference frame convention. This function dictates that the c[0]th cell vector is colinear with the c[0]th axis, the c[1]th cell vector is perpendicular to the c[2]th axis, and the c[2]th cell vector is defined to give a right-handed coordinate system. In common crystallographic shorthand, x[c[0]] // a[c[0]], x[c[2]] // a[c[2]]^* (i.e. perpendicular to a[c[0]] and a[c[1]]).

Returns

The three vectors defining the parallelopiped cell [m].

Return type

numpy.array (2D)

burnman.**cell_vectors_to_parameters**(*vectors*, *frame_convention*, *rotate=False*)

Converts unit cell vectors to cell parameters.

Parameters

- **vectors** (*numpy.array (2D)*) – The three vectors defining the parallelopiped cell [m].
- **frame_convention** (*list of three integers*) – A list (c) defining the reference frame convention. This function dictates that the c[0]th cell vector is colinear with the c[0]th axis, the c[1]th cell vector is perpendicular to the c[2]th axis, and the c[2]th cell vector is defined to give a right-handed coordinate system. In common crystallographic shorthand, x[c[0]] // a[c[0]], x[c[2]] // a[c[2]]^* (i.e. perpendicular to a[c[0]] and a[c[1]]).
- **rotate** (*bool*) – If True, the input vectors are rotated to satisfy the requested frame convention. If False, the input vectors are assumed to already satisfy the requested frame convention.

Returns

An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Return type

numpy.array (1D)

20.7 Composites

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

Bases: *Material*

The composite class in BurnMan is designed to calculate the properties of multi-phase assemblages. Each phase within the composite is represented by an instance of the Material class (typically a Mineral or Solution). The composite class calculates the bulk properties of the assemblage based on the properties and proportions of the individual phases.

All of the **thermodynamic** properties (Gibbs free energy, enthalpy, entropy, volume, heat capacity etc.) are calculated by assuming that any changes in pressure or temperature of the composite as a whole are reflected equally in all of the individual phases, but that no reactions occur either within or between phases. For a composite that calculates properties assuming some or all reactions between phases, can take place ion the timescales of changes in pressure and temperature, see *Relaxed Composite Class*.

The **elastic** properties (bulk and shear moduli, seismic velocities etc.) of a composite material on short timescales are often not well-approximated by assuming homogeneous changes in pressure and temperature. This is because differential stresses can develop between phases due to their differing elastic and thermal properties. To account for this, the composite class allows the user to specify an averaging scheme for the elastic moduli of the composite. The default averaging scheme is Voigt-Reuss-Hill, which is appropriate for many applications. Other *Averaging Schemes* are available.

Example:

```
from burnman import Composite
from burnman.minerals import SLB2011

# Create a composite made of 70% olivine and 30% orthopyroxene
phases = [SLB2011.mg_fe_olivine(), SLB2011.orthopyroxene()]
fractions = [0.7, 0.3]  # molar fractions
composite = Composite(phases, fractions, fraction_type='molar', name=
↪'Mantle composite')

# Set the compositions of the solution phases
# Endmembers of olivine are forsterite and fayalite
# Endmembers of orthopyroxene are enstatite, ferrosilite, Mg-tschermaks and␣
↪Orthodiopside
composite.phases[0].set_composition([0.9, 0.1])  # Fo90
composite.phases[1].set_composition([0.8, 0.2, 0.0, 0.0])  # En80

# Set the state of the composite (1 GPa, 1500 K)
composite.set_state(1e9, 1500)

# Change the fractions of the phases
composite.set_fractions([0.6, 0.4])

# Change the averaging scheme to Hashin-Shtrikman lower bound
```

(continues on next page)

(continued from previous page)

```
composite.set_averaging_scheme("HashinShtrikmanLower")
```

Parameters

- **phases** (list of *burnman.Material*) – List of phases.
- **fractions** (*list of floats*) – molar or mass fraction for each phase.
- **fraction_type** (*str*) – 'molar' or 'mass' (optional, 'molar' as standard) specify whether molar or mass fractions are specified.
- **name** (*str*) – Name of the composite (optional).

property name

Human-readable name of this material.

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

set_fractions(*fractions*, *fraction_type='molar'*)

Change the fractions of the phases of this Composite. Resets cached properties

Parameters

- **fractions** – list or numpy array of floats molar or mass fraction for each phase.
- **fraction_type** – 'molar' or 'mass' specify whether molar or mass fractions are specified.

set_method(*method*)

set the same equation of state method for all the phases in the composite

set_averaging_scheme(*averaging_scheme*)

Set the averaging scheme for the moduli in the composite. Default is set to VoigtReussHill, when Composite is initialized.

set_state(*pressure*, *temperature*)

Update the material to the given pressure [Pa] and temperature [K].

debug_print(*indent=''*)

Print a human-readable representation of this Material.

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

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

Return type
tuple

to_string()
return the name of the composite

property phase_volumes
Returns total volumes of each phase in the composite

property phase_masses
Returns total masses of each phase in the composite

property formula
Returns extensive 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_reuss
Returns isothermal bulk modulus of the composite [Pa] Aliased with self.K_T

property thermal_expansivity

Returns thermal expansion coefficient of the composite [1/K] Aliased with self.alpha

property molar_heat_capacity_p

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

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the mineral [Pa]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isentropically compressed, since heat exchange between phases may be required to maintain the same temperature in all phases.

Aliased with self.K_S

property isothermal_compressibility_reuss

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

property isentropic_compressibility_reuss

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

property molar_heat_capacity_v

Returns molar_heat capacity at constant volume of the composite [J/K/mol]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isochorically heated, since differential expansion between phases may be required to maintain the same pressure in all phases.

Aliased with self.C_v

property grueneisen_parameter

Returns grueneisen parameter of the composite [unitless]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium.

Aliased with self.gr

property effective_isentropic_bulk_modulus

Returns the effective isotropic, isentropic bulk modulus of the composite [Pa] Aliased with self.K_eff

property effective_shear_modulus

Returns the effective isotropic shear modulus of the mineral [Pa] Aliased with self.G_eff

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 endmember_partial_gibbs

Returns the partial Gibbs energies for all the endmember minerals in the Composite

property endmember_partial_entropies

Returns the partial entropies for all the endmember minerals in the Composite

property endmember_partial_volumes

Returns the partial volumes for all the endmember minerals in the Composite

property reaction_affinities

Returns the affinities corresponding to each reaction in reaction_basis

property equilibrated

Returns True if the reaction affinities are all zero within a given tolerance given by self.equilibrium_tolerance.

set_components(*components*)

Sets the components and components_array attributes of the composite material. The components attribute is a list of dictionaries containing the chemical formulae of the components. The components_array attribute is a 2D numpy array describing the linear transformation between endmember amounts and component amounts.

The components do not need to be linearly independent, not do they need to form a complete basis set for the composite. However, it must be possible to obtain the composition of each component from a linear sum of the endmember compositions of the composite. For example, if the composite was composed of MgSiO3 and Mg2SiO4, SiO2 would be a valid component, but Si would not. The method raises an exception if any of the chemical potentials are not defined by the assemblage.

Parameters

components (*list of dictionaries*) – List of formulae of the components.

chemical_potential(*components=None*)

Returns the chemical potentials of the currently defined components in the composite. Raises an exception if the assemblage is not equilibrated.

Parameters

components (*list of dictionaries*) – List of formulae of the desired components. If not specified, the method uses the components specified by a previous call to set_components.

Returns

The chemical potentials of the desired components in the equilibrium composite.

Return type

numpy.array of floats

property stoichiometric_matrix

An sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property reaction_basis_as_strings

Returns a list of string representations of all the reactions in reaction_basis.

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts)

property dependent_element_indices

The element indices not included in the independent list.

property reduced_stoichiometric_array

The stoichiometric array including only the independent elements

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the composite.

property endmember_formulae

A list of the formulae in the composite.

property endmember_names

A list of the endmember names contained in the composite. Mineral names are returned as given in Mineral.name. Solution endmember names are given in the format *Mineral.name in Solution.name*.

property endmembers_per_phase

A list of integers corresponding to the number of endmembers stored within each phase.

property elements

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

property n_endmembers

Returns the number of endmembers in the composite.

property n_elements

Returns the total number of distinct elements which might be in the composite.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

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

Note

Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

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

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral.

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

Returns

Shear modulus in [Pa].

Return type

float

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

property volume
Extensive volume of the material (m^3).

class burnman.**RelaxedComposite**(*composite*, *relaxation_vectors*)

Bases: *Composite*

A class implementing a relaxed Composite mineral assemblage. Unlike a standard Composite, the set_state() method of this class uses the *burnman.equilibrate* function to find the equilibrium phase fractions and compositions at given pressure and temperature. Once equilibrated, the thermodynamic properties of the relaxed composite can be queried in the same way as for a standard Composite.

One additional feature of this class is that properties related to the second derivatives of the Gibbs energy (e.g. compressibility, thermal expansivity, heat capacity) can be calculated taking into account relaxation of one or more user-defined isochemical reaction vectors via the relaxation_vectors argument to the constructor. If complete relaxation is desired, the user can pass relaxation_vectors=composite.reaction_basis. More limited relaxation can be specified by passing a linear subset of the reaction basis vectors.

For example, on short timescales,a composite of olivine and wadsleyite might be able to exchange Mg and Fe, even if no grain growth or reduction can occur. In this case, a single vector representing the exchange of Mg and Fe between the two minerals can be used to relax the composition at given P and T: relaxation_vectors=[[1, -1, -1, 1]], where the first two entries correspond to olivine (fo, fa) and the second two to wadsleyite (mwd, fwd). This is a subset of the full reaction basis, which would be: [[1, 0, -1, 0], [0, 1, 0, -1]].

Example:

```
from burnman import RelaxedComposite

# Create a standard Composite
# This is also where you would set the composition of
```

(continues on next page)

(continued from previous page)

```
# the individual phases if desired.
unrelaxed_composite = Composite(...) # see Composite documentation

# Create a RelaxedComposite that reacts quickly relative to
# changes in pressure and temperature
composite = RelaxedComposite(
    unrelaxed_composite,
    relaxation_vectors=unrelaxed_composite.reaction_basis
)

# Set the state of the relaxed composite at 10 GPa and 1500 K
# This will equilibrate the composite at its current bulk
# composition.
composite.set_state(1.0e10, 1500.0)

# Now do the same but specifying a different bulk composition
bulk_composition = {'Mg': 0.8, 'Fe': 0.2, 'Si': 1.0, 'O': 4.0}
composite.set_state(1.0e10, 1500.0, bulk_composition=bulk_composition)

# Query thermodynamic properties as usual
K_T = composite.isothermal_bulk_modulus_reuss
```

Parameters

- **composite** (*burnman.Composite*) – The unrelaxed Composite material.
- **relaxation_vectors** (*list of lists or 2D array*) – A list of isochemical relaxation vectors. Each vector should be a list or array with length equal to the number of endmembers in the composite. The number of vectors determines the number of degrees of freedom available for relaxation.

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

set_state(*pressure*, *temperature*, *bulk_composition=None*)

Sets the state of the composite. Also relaxes the structure parameters if set_composition() has already been used and if the relaxed argument has been set to True.

Parameters

- **pressure** (*float*) – The pressure of the solution [Pa]
- **temperature** (*float*) – The temperature of the solution [K]

- **bulk_composition** (*dict*) – The bulk composition of the composite in terms of elemental amounts. If None, the bulk composition stored in the unrelaxed composite is used.

copy_state_from_unrelaxed()

Copies the state (P, T, molar_fractions) from the unrelaxed composite to the relaxed composite. This can be useful if the unrelaxed composite has been modified externally and the relaxed composite needs to be updated to match.

property dqdz_relaxed

The change of the structure parameters with respect to pressure and temperature that minimizes the Gibbs energy.

property isothermal_compressibility_reuss

(copied from isothermal_compressibility_reuss):

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

property thermal_expansivity

(copied from thermal_expansivity):

Returns thermal expansion coefficient of the composite [1/K] Aliased with self.alpha

property molar_heat_capacity_p

(copied from molar_heat_capacity_p):

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

property isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the composite [Pa] Aliased with self.K_T

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property bulk_sound_velocity

Returns bulk sound speed of the composite [m/s] Aliased with self.v_phi

chemical_potential(*components=None*)

Returns the chemical potentials of the currently defined components in the composite. Raises an exception if the assemblage is not equilibrated.

Parameters

components (*list of dictionaries*) – List of formulae of the desired components. If not specified, the method uses the components specified by a previous call to set_components.

Returns

The chemical potentials of the desired components in the equilibrium composite.

Return type

numpy.array of floats

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the composite.

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

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 effective_isentropic_bulk_modulus

Returns the effective isotropic, isentropic bulk modulus of the composite [Pa] Aliased with self.K_eff

property effective_shear_modulus

Returns the effective isotropic shear modulus of the mineral [Pa] Aliased with self.G_eff

property elements

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

property endmember_formulae

A list of the formulae in the composite.

property endmember_names

A list of the endmember names contained in the composite. Mineral names are returned as given in Mineral.name. Solution endmember names are given in the format *Mineral.name in Solution.name*.

property endmember_partial_entropies

Returns the partial entropies for all the endmember minerals in the Composite

property endmember_partial_gibbs

Returns the partial Gibbs energies for all the endmember minerals in the Composite

property endmember_partial_volumes

Returns the partial volumes 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 enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

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*, *molar_fractions=None*)

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

Parameters

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

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property formula

Returns extensive chemical formula of the composite

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the composite [unitless]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium.

Aliased with self.gr

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the mineral [Pa]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isentropically compressed, since heat exchange between phases may be required to maintain the same temperature in all phases.

Aliased with self.K_S

property isentropic_compressibility_reuss

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

property isentropic_thermal_gradient

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

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_v

Returns molar_heat capacity at constant volume of the composite [J/K/mol]

Note: This function implicitly assumes that all of the individual phases are, and remain, in thermal and mechanical equilibrium. A corollary is that the individual phases may not be isochorically heated, since differential expansion between phases may be required to maintain the same pressure in all phases.

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 phase_masses

Returns total masses of each phase in the composite

property phase_volumes

Returns total volumes of each phase in the composite

property pressure

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

> **Note**
>
> Aliased with *P()*.

Returns

Pressure in [Pa].

Return type

float

print_minerals_of_current_state()

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

property reaction_affinities

Returns the affinities corresponding to each reaction in reaction_basis

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property reaction_basis_as_strings

Returns a list of string representations of all the reactions in reaction_basis.

property reduced_stoichiometric_array

The stoichiometric array including only the independent elements

reset()

Resets all cached material properties.

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

property rho

Alias for *density()*

set_averaging_scheme(*averaging_scheme*)

Set the averaging scheme for the moduli in the composite. Default is set to VoigtReussHill, when Composite is initialized.

set_components(*components*)

Sets the components and components_array attributes of the composite material. The components attribute is a list of dictionaries containing the chemical formulae of the components. The components_array attribute is a 2D numpy array describing the linear transformation between endmember amounts and component amounts.

The components do not need to be linearly independent, not do they need to form a complete basis set for the composite. However, it must be possible to obtain the composition of each component from a linear sum of the endmember compositions of the composite. For example, if the composite was composed of MgSiO3 and Mg2SiO4, SiO2 would be a valid component, but Si would not. The method raises an exception if any of the chemical potentials are not defined by the assemblage.

Parameters

components (*list of dictionaries*) – List of formulae of the components.

set_fractions(*fractions*, *fraction_type='molar'*)

Change the fractions of the phases of this Composite. Resets cached properties

Parameters

- **fractions** – list or numpy array of floats molar or mass fraction for each phase.
- **fraction_type** – 'molar' or 'mass' specify whether molar or mass fractions are specified.

set_method(*method*)

set the same equation of state method for all the phases in the composite

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

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding method. To provide more context, even if a mineral is being evaluated with a P(V, T) equation of state, there might be a property modifier G_mod(P, T) added on top - which might then introduce a pressure dependent V_mod(P, T). We must therefore find the pressure P(V, T) iteratively: V = V_base(P, T) + V_mod(P, T), even if there is a function for P_base(V_base, T).

This function is overloaded by the Mineral class.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the mineral.

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

Returns
Shear modulus in [Pa].

Return type
float

property shear_wave_velocity

Returns shear wave speed of the composite [m/s] Aliased with self.v_s

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_matrix

An sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property temperature

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

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

return the name of the composite

unroll()

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

> **Note**
>
> Needs to be implemented in derived classes.

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

Return type
tuple

property v_p
Alias for *p_wave_velocity()*

property v_phi
Alias for *bulk_sound_velocity()*

property v_s
Alias for *shear_wave_velocity()*

property volume
Extensive volume of the material (m^3).

20.8 Calibrants

class burnman.**Calibrant**(*calibrant_function*, *calibrant_function_return_type*, *params*)

Bases: object

The base class for a pressure calibrant material.

Parameters

- **calibrant_function** (*function*) – A function that takes either pressure, temperature and a params object as arguments, returning the volume, or takes volume, temperature and a params object, returning the pressure.
- **calibrant_function_return_type** (*str*) – The return type of the calibrant function. Valid values are 'pressure' or 'volume'.
- **params** (*dictionary*) – A dictionary containing the parameters required by the calibrant function.

pressure(*volume*, *temperature*, *VT_covariance=None*)

Returns the pressure of the calibrant as a function of volume, temperature and (optionally) a volume-temperature variance-covariance matrix.

Parameters

- **volume** (*float*) – The volume of the calibrant [m^3/mol].
- **temperature** (*float*) – The temperature of the calibrant [K].

- **VT_covariance** (*2x2 numpy.array*) – The volume-temperature variance-covariance matrix [optional].

Returns
The pressure of the calibrant [Pa] and the pressure-volume-temperature variance-covariance matrix if PT_covariance is provided.

Return type
float, tuple of a float and a numpy.array (3x3)

volume(*pressure*, *temperature*, *PT_covariance=None*)

Returns the volume of the calibrant as a function of pressure, temperature and (optionally) a pressure-temperature variance-covariance matrix.

Parameters

- **pressure** (*float*) – The pressure of the calibrant [Pa].
- **temperature** (*float*) – The temperature of the calibrant [K].
- **PT_covariance** (*2x2 numpy.array*) – The pressure-temperature variance-covariance matrix [optional].

Returns
The volume of the calibrant [m^3/mol] and the volume-pressure-temperature variance-covariance matrix if VT_covariance is provided.

Return type
float, tuple of a float and a numpy.array (3x3)

20.9 Equation of State Classes and Functions

- *Isothermal Equations of State*
  - *Base class*
  - *birch murnaghan*
  - *macaw*
  - *modified tait*
  - *morse potential*
  - *murnaghan*
  - *reciprocal kprime*
  - *spock*
  - *vinet*
- *Thermal Equations of State*
  - *Base class*

    - *aa*
    - *brosh calphad*
    - *cork*
    - *dks liquid*
    - *dks solid*
    - *hp*
    - *mie grueneisen debye*
    - *modular mie grueneisen debye*
    - *slb*
- *Other Equation of State classes*
    - *anharmonic debye*
    - *anharmonic prefactor models*
    - *anharmonic thermal models*
    - *debye temperature models*
- *Equation of State Functions*
    - *anharmonic thermal models*
    - *birch murnaghan*
    - *bukowinski electronic*
    - *debye*
    - *debye temperature models*
    - *einstein*
    - *macaw*
    - *modified tait*
    - *morse potential*
    - *murnaghan*
    - *property modifiers*
    - *reciprocal kprime*
    - *vinet*

20.9.1 Isothermal Equations of State

20.9.1.1 Base class

class burnman.eos.**IsothermalEquationOfState**

Bases: *EquationOfState*

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

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

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

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

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

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

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

Return type

float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Gibbs energy of the mineral $[J/mol]$.

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Isothermal bulk modulus of the mineral. $[Pa]$

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

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

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

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

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Shear modulus of the mineral. $[Pa]$

Return type
float

validate_parameters(*params*)

The params object is just a dictionary associating mineral physics parameters for the equation of state. Different equation of states can have different parameters, and the parameters may have ranges of validity. The intent of this function is twofold. First, it can check for the existence of the parameters that the equation of state needs, and second, it can check whether the parameters have reasonable values. Unreasonable values will frequently be due to unit issues (e.g., supplying bulk moduli in GPa instead of Pa). In the base class this function does nothing, and an equation of state is not required to implement it. This function will not return anything, though it may raise warnings or errors.

Parameters
params (*dict*) – Dictionary containing material parameters required by the equation of state.

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

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

Return type
float

20.9.1.2 birch murnaghan

class burnman.eos.birch_murnaghan.**BM3**

Bases: *BirchMurnaghanBase*

The third order Birch-Murnaghan isothermal equation of state. The negative finite-strain (or compression) is defined as

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

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:

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

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

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

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.

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy	J/mol
`P_0`	Reference pressure	Pa
`V_0`	Reference volume	m^3
`K_0`	Reference isothermal bulk modulus	Pa
`Kprime_0`	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
`G_0`	Reference shear modulus	Pa
`Gprime_0`	Pressure derivative of the shear modulus at the reference state	Dimensionless

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

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

Return type
float

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

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

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

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

validate_parameters(*params*)

Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

class burnman.eos.birch_murnaghan.**BM3Shear2**

Bases: *BirchMurnaghanBase*

The third order Birch-Murnaghan isothermal equation of state with second order expansion for the shear modulus. Do not use this unless you have a good reason to.

The negative finite-strain (or compression) is defined as

$$f = \frac{1}{2}\left[\left(\frac{V}{V_0}\right)^{-2/3} - 1\right]$$

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).

For an isotropic material one obtains for the pressure, isothermal bulk modulus, and shear modulus:

$$P = 3K_0 f\left(1+2f\right)^{5/2}\left[1+\frac{3}{2}\left(K_0'-4\right)f\right],$$

$$K_T = \left(1+2f\right)^{5/2}\Big[K_0 + \left(3K_0K_0' - 5K_0\right)f + \frac{27}{2}\left(K_0K_0' - 4K_0\right)f^2\Big],$$

$$G = \left(1+2f\right)^{5/2}\Big[G_0 + \left(3K_0G_0' - 5G_0\right)f\Big].$$

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.

Parameter	Description	Units
F_0	Reference Helmholtz free energy	J/mol
P_0	Reference pressure	Pa
V_0	Reference volume	m^3
K_0	Reference isothermal bulk modulus	Pa
Kprime_0	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
G_0	Reference shear modulus	Pa
Gprime_0	Pressure derivative of the shear modulus at the reference state	Dimensionless

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type

float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Internal energy of the mineral $[J/mol]$.

Return type

float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns shear modulus G of the mineral. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)
Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

validate_parameters(*params*)
Check for existence and validity of the parameters

volume(*pressure*, *temperature*, *params*)
Returns volume $[m^3]$ as a function of pressure $[Pa]$.

class burnman.eos.birch_murnaghan.**BM4**
Bases: *BirchMurnaghanBase*

Base class for the isothermal Birch Murnaghan equation of state. This is fourth order in strain, and has no temperature dependence.

Note that unlike the third order Birch-Murnaghan equation of state, the shear modulus is not defined for this equation of state.

Parameter	Description	Units
F_0	Reference Helmholtz free energy	J/mol
P_0	Reference pressure	Pa
V_0	Reference volume	m^3
K_0	Reference isothermal bulk modulus	Pa
Kprime_0	Pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless
Kprime_prime_0	Second pressure derivative of the isothermal bulk modulus at the reference state	Dimensionless

volume(*pressure*, *temperature*, *params*)
Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Grueneisen parameter of the mineral. $[unitless]$

Return type

float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Helmholtz energy of the mineral $[J/mol]$.

Return type

float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

class burnman.eos.birch_murnaghan.**BirchMurnaghanBase**

Bases: *IsothermalEquationOfState*

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.BM3Shear2 and burnman.birch_murnaghan.BM3.

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)
Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.3 macaw

class burnman.eos.macaw.**MACAW**

Bases: *IsothermalEquationOfState*

Class for the MACAW equation of state detailed in [LA22].

This equation of state has no temperature dependence.

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of the bulk modulus at reference pressure.	Dimensionless
Kprime_inf	Infinite pressure derivative of the bulk modulus.	Dimensionless

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

volume(*pressure*, *temperature*, *params*)

Get the Vinet volume at a reference temperature for a given pressure $[Pa]$. Returns molar volume in $[m^3]$

pressure(*temperature*, *volume*, *params*)

Returns pressure $[Pa]$ as a function of volume $[m^3]$.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

validate_parameters(*params*)

Check for existence and validity of the parameters. The value for K'_∞ is thermodynamically bounded between 5/3 and K'_0 [SD04].

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.4 modified tait

class burnman.eos.modified_tait.**MT**

Bases: *IsothermalEquationOfState*

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].

$$\frac{V_{P,T}}{V_{1bar,298K}} = 1 - a(1 - (1 + bP)^{-c}),$$
$$a = \frac{1 + K_0'}{1 + K_0' + K_0K_0''},$$
$$b = \frac{K_0'}{K_0} - \frac{K_0''}{1 + K_0'},$$
$$c = \frac{1 + K_0' + K_0K_0''}{K_0'^2 + K_0' - K_0K_0''}$$

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus.	Dimensionless
Kdprime_0	Second pressure derivative of bulk modulus.	Pa^{-1}

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_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T of the mineral. $[Pa]$.

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.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type

float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Internal energy of the mineral $[J/mol]$.

Return type

float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.5 morse potential

class burnman.eos.morse_potential.**Morse**

Bases: *IsothermalEquationOfState*

Class for the isothermal Morse Potential equation of state detailed in [StaceyBrennanIrvine81]. This equation of state has no temperature dependence.

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`P_0`	Reference pressure.	Pa
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of bulk modulus at zero pressure.	Dimensionless

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* [m^3].
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. [kg/m^3]

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the equation of state [K].
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* [m^3].
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral [J/mol].

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)
Entropy of the equation of state ([$J/K/mol$]). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the equation of state [K].
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* [m^3].
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. [$unitless$]

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.6 murnaghan

class burnman.eos.murnaghan.**Murnaghan**

Bases: *IsothermalEquationOfState*

The isothermal Murnaghan equation of state, as described in [Mur44].

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus.	Dimensionless

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$ Currently not included in the Murnghan EOS, so omitted.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.7 reciprocal kprime

class burnman.eos.reciprocal_kprime.**RKprime**

Bases: *IsothermalEquationOfState*

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.

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus at reference conditions.	Dimensionless
Kprime_inf	Pressure derivative of bulk modulus at infinite pressure.	Dimensionless

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_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters. The value for K'_∞ is thermodynamically bounded between 5/3 and K'_0 [SD04].

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type

float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Internal energy of the mineral $[J/mol]$.

Return type

float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.8 spock

class burnman.eos.spock.**SPOCK**

Bases: *IsothermalEquationOfState*

Class for the Scaled Power Of Compression K-prime equation of state. This equation is derived from the assumption that

$$K' = b\left(\frac{V}{V_0}\right)^a$$

This equation of state is described in [Myh25a], but was originally derived by [WMTZ15].

Parameter	Description	Units
`F_0`	Reference Helmholtz free energy.	J/mol
`P_0`	Reference pressure.	Pa
`V_0`	Reference volume.	m^3
`K_0`	Reference bulk modulus.	Pa
`Kprime_0`	Pressure derivative of bulk modulus at zero pressure.	Dimensionless
`Kdprime_0`	Second pressure derivative of bulk modulus at zero pressure.	Pa^{-1}
`Kprime_inf`	Infinite pressure limit of the pressure derivative of bulk modulus.	Dimensionless

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

volume(*pressure*, *temperature*, *params*)

Returns volume at a given pressure $[Pa]$ in $[m^3]$

pressure(*temperature*, *volume*, *params*)

Returns pressure $[Pa]$ as a function of volume $[m^3]$.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$ This equation of state is athermal, so the function returns a very large number.

validate_parameters(*params*)

Check for existence and validity of the parameters. The value for K'_∞ is thermodynamically bounded between 5/3 and K'_0 [SD04].

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.1.9 vinet

class burnman.eos.vinet.**Vinet**

Bases: *IsothermalEquationOfState*

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].

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
P_0	Reference pressure.	Pa
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of bulk modulus.	Dimensionless

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus G of the mineral. $[Pa]$ Not included in the Vinet EOS, so returns 0.

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy $\mathcal{G}$ of the mineral. $[J/mol]$

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Entropy of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so the entropy arbitrarily returns a value of zero.

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Isobaric molar heat capacity of the equation of state ($[J/K/mol]$). This equation of state is isothermal, so this function arbitrarily returns a very small number.

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Thermal expansivity of the equation of state ($[1/K]$). This equation of state is isothermal, so this function arbitrarily returns a value of zero.

20.9.2 Thermal Equations of State

20.9.2.1 Base class

class burnman.eos.**EquationOfState**

Bases: object

This class defines the interface for an equation of state that a mineral uses to determine its properties at a given P, T. In order define a new equation of state, you should define these functions.

All functions should accept and return values in SI units.

In general these functions are functions of pressure, temperature, and volume, as well as a "params" object, which is a Python dictionary that stores the material parameters of the mineral, such as reference volume, Debye temperature, reference moduli, etc.

The functions for volume and density are just functions of temperature, pressure, and "params"; after all, it does not make sense for them to be functions of volume or density.

volume(*pressure*, *temperature*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Molar volume of the mineral $[m^3]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Isothermal bulk modulus of the mineral. $[Pa]$

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Shear modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant pressure of the mineral. $[J/K/mol]$

Return type
float

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Thermal expansivity of the mineral. $[1/K]$

Return type
float

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Gibbs energy of the mineral $[J/mol]$.

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Entropy of the mineral $[J/K/mol]$.

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

validate_parameters(*params*)

The params object is just a dictionary associating mineral physics parameters for the equation of state. Different equation of states can have different parameters, and the parameters may have ranges of validity. The intent of this function is twofold. First, it can check for the existence of the parameters that the equation of state needs, and second, it can check whether the parameters have reasonable values. Unreasonable values will frequently be due to unit issues (e.g., supplying bulk moduli in GPa instead of Pa). In the base class this function does nothing, and an equation

of state is not required to implement it. This function will not return anything, though it may raise warnings or errors.

Parameters
params (*dict*) – Dictionary containing material parameters required by the equation of state.

20.9.2.2 aa

class burnman.eos.aa.**AA**

Bases: *EquationOfState*

Class for the liquid metal EOS detailed in [AndersonAhrens94].

This equation of state is complicated because there is not a single set of independent variables.

The equation of state is based on a reference isentrope passing through a defined volume and entropy point. Internal energy (E) at a given volume is calculated along this isentrope using a fourth order BM EoS (V_0, KS, KS', KS'').

The temperature along the isentrope is calculated via integration of the Grueneisen parameter:

$\gamma = \partial(\ln T)/\partial(\ln \rho)|_S$

which gives:

$T_S/T_0 = \exp(\int(\gamma/\rho)d\rho)$

Finally, the internal energy away from the reference isentrope is calculated as a function of temperature, using an expression for the isochoric heat capacity as a function of volume and temperature.

In this BurnMan implementation, the Helmholtz energy is used as the natural potential, with volume and temperature as the natural variables.

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, the thermal pressure expression is inconsistent with the equation of state away from the principal isentrope.

Note: the expression for Λ (Appendix C) does not reproduce Figure 5. We assume that the figure is correct, and that the correct expression has the form: $F(-325.23 + 302.07(\rho/\rho_0) + 30.45(\rho/\rho_0)^{0.4})$.

Parameter	Description	Units
E_0	Reference internal energy.	J/mol
V_0	Reference volume.	m^3
K_S	Reference isentropic bulk modulus.	Pa
Kprime_S	First pressure derivative of the isentropic bulk modulus.	unitless
Kprime_prime_S	Second pressure derivative of the isentropic bulk modulus.	Pa^{-1}
T_0	Reference temperature on the principal isentrope.	K
S_0	Reference entropy on the principal isentrope.	J/K/mol
n	Number of atoms per formula unit.	unitless
grueneisen_0	Reference Gruneisen parameter on the principal isentrope.	unitless
grueneisen_n	Volume dependence of the Gruneisen parameter on the principal isentrope.	unitless
grueneisen_prim	First pressure derivative of the Gruneisen parameter on the principal isentrope.	Pa^{-1}
theta	Debye temperature on the principal isentrope.	K
a	Electronic heat capacity parameter A (length 2 array).	unitless
b	Electronic heat capacity parameter B (length 2 array).	unitless
Theta	Volume dependence parameters for electronic heat capacity (length 2 array).	unitless
lmda	Potential heat capacity parameter lambda (length 3 array).	unitless
xi_0	Reference xi parameter.	unitless
F	Reference F parameter (length 2 array).	unitless

entropy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Entropy of the mineral $[J/K/mol]$.

Return type

float

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

volume(*pressure*, *temperature*, *params*)

Returns molar volume. $[m^3]$

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns the thermal expansivity. $[1/K]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol] F + PV

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$ Zero for a liquid

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.2.3 brosh calphad

class burnman.eos.brosh_calphad.**BroshCalphad**

Bases: *EquationOfState*

Class for the high pressure CALPHAD equation of state by [BMS07].

This equation of state includes contributions from cold compression, quasiharmonic lattice vibrations, and thermal excitation of electrons.

Parameter	Description	Units
gibbs_coefficie	Coefficients for the piecewise polynomial fit to the Gibbs free energy at 1 bar. Format is a list of lists, where each sublist contains the maximum temperature for that segment and the coefficients in the following order: [const, T, T*ln(T), T^(-1), T^(-2), T^(-3), T^(-9), T^2, T^3, T^4, T^7, T^(1/2), ln(T)]	varies
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
n	Number of atoms per formula unit.	Dimensionless
a	Parameter array for cold compression (length 4 array).	Dimensionless
b	Parameter array for thermal excitation (length 2 array).	Dimensionless
c	[OPTIONAL] Parameter array for cold compression (length 4 array). If you do not know how to calculate it, do not add it to the parameters dictionary. It will be calculated from the other input parameters.	Dimensionless
delta	Parameter array for thermal excitation (length 2 array).	Dimensionless
theta_0	Reference Debye temperature.	K
grueneisen_0	Reference Grüneisen parameter.	Dimensionless

volume(*pressure*, *temperature*, *params*)

Returns volume $[m^3]$ as a function of pressure $[Pa]$.

dvolume_dP(*pressure*, *temperature*, *params*)

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns the isothermal bulk modulus K_T $[Pa]$ as a function of pressure $[Pa]$, temperature $[K]$ and volume $[m^3]$.

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns the shear modulus G of the mineral. $[Pa]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy of the mineral. $[J/mol]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the molar entropy of the mineral. $[J/K/mol]$

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the molar isobaric heat capacity $[J/K/mol]$. For now, this is calculated by numerical differentiation.

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns the volumetric thermal expansivity $[1/K]$. For now, this is calculated by numerical differentiation.

calculate_transformed_parameters(*params*)

This function calculates the "c" parameters of the [BMS07] equation of state.

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.2.4 cork

class burnman.eos.cork.**CORK**

Bases: *EquationOfState*

Class for the CoRK equation of state detailed in [HP91]. The CoRK EoS is a simple virial-type extension to the modified Redlich-Kwong (MRK) equation of state. It was designed to compensate for the tendency of the MRK equation of state to overestimate volumes at high pressures and accommodate the volume behaviour of coexisting gas and liquid phases along the saturation curve.

$$
\begin{aligned}
V &= \frac{RT}{P} + c_1 - \frac{c_0 RT^{0.5}}{(RT + c_1 P)(RT + 2c_1 P)} + c_2 P^{0.5} + c_3 P, \\
c_0 &= c_{0,0} T_c^{2.5}/P_c + c_{0,1} T_c^{1.5}/P_c T, \\
c_1 &= c_{1,0} T_c / P_c, \\
c_2 &= c_{2,0} T_c / P_c^{1.5} + c_{2,1}/P_c^{1.5} T, \\
c_3 &= c_{3,0} T_c / P_c^2 + c_{3,1}/P_c^2 T
\end{aligned}
$$

where $c_{i,j}$ are the CoRK parameters, T_c is the critical temperature, and P_c is the critical pressure.

Parameter	Description	Units
H_0	Reference enthalpy.	J/mol
S_0	Reference entropy.	J/(mol K)
Cp	Heat capacity parameters (length 4 array).	J/(mol K)
T_0	Reference temperature.	K
P_0	Reference pressure.	Pa
cork_params	CoRK parameters (length 4x2 array).	varies
cork_T	Critical temperature.	K
cork_P	Critical pressure.	Pa

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_reuss(*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.

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_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure at the pressure, temperature, and volume [J/K/mol]

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

20.9.2.5 dks liquid

class burnman.eos.dks_liquid.**DKS_L**

Bases: *EquationOfState*

Base class for the finite strain liquid equation of state detailed in [deKokerKarkiStixrude13] (supplementary materials).

This equation of state includes contributions from ideal gas (translational and electronic), electronic, and excess (bonding) components to the thermodynamic properties.

The parameters required for this equation of state do not all correspond to natural variables; please refer to [deKokerKarkiStixrude13] for details.

Parameter	Description	Units
V_0	Reference volume.	m^3
T_0	Reference temperature.	K
O_theta		
O_f		
m		
a		
zeta_0	Reference electronic strength parameter.	
xi	Power law exponent for the volume dependence on zeta.	Unitless
Tel_0	Reference temperature for electronic contributions.	K
eta	Power law exponent for the volume dependence on the electronic temperature.	Unitless
formula	Chemical formula of the substance.	Dict[str, float]
el_V_0	Reference volume for electronic contributions.	m^3
spin_a	Description of spin_a.	Length 2 list describing spin_a.
spin_b	Length 4 list describing spin_b.	Units of spin_b.

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type

float

volume(*pressure*, *temperature*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Molar volume of the mineral $[m^3]$.

Return type
float

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)
Returns isothermal bulk modulus $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns shear modulus. $[Pa]$ Zero for fluids

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_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]

validate_parameters(*params*)
Check for existence and validity of the parameters

density(*volume*, *params*)
Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.2.6 dks solid

class burnman.eos.dks_solid.**DKS_S**

Bases: *EquationOfState*

Base class for the finite strain solid equation of state detailed in [deKokerKarkiStixrude13] (supplementary materials).

This equation of state is a somewhat modified version of that described in [SLB05].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
E_0	Reference internal energy.	J/mol
S_0	Reference entropy.	J/K/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of the reference bulk modulus.	unitless
grueneisen_0	Reference Grüneisen parameter.	unitless
q_0	Volume derivative of the Grüneisen parameter.	unitless
eta_s_0	Isotropic shear strain derivative of the Grüneisen parameter.	unitless
Cv	Constant volume heat capacity.	J/K/mol

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]

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)
Returns isothermal bulk modulus $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)
Returns shear modulus. $[Pa]$

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_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]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.2.7 hp

class burnman.eos.hp.**HP98**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
H_0	Enthalpy at the reference state.	J/mol
S_0	Entropy at the reference state.	J/(mol K)
Cp	Heat capacity coefficients at the reference pressure (length 4 list).	Various
V_0	Volume at the reference state.	m^3/mol
K_0	Isothermal bulk modulus at the reference state.	Pa
Kprime_0	Pressure derivative of bulk modulus at the reference state.	Dimensionless
a_0	Thermal expansivity at reference state.	1/K
dKdT_0	Temperature derivative of bulk modulus at reference state.	Pa/K

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

isothermal_bulk_modulus_reuss(*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.

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

class burnman.eos.hp.**HP_TMT**

Bases: *EquationOfState*

Base class for the thermal equation of state based on the isothermal modified Tait equation of state (class MT), as described in [HollandPowell11].

$$\mathcal{G}(P,T) = \mathcal{H}_{\text{1 bar, T}} - T\mathcal{S}_{\text{1 bar, T}} + \int_{\text{1 bar}}^{P} V(P,T)\,dP,$$

$$\mathcal{H}_{\text{1 bar, T}} = \Delta_f \mathcal{H}_{\text{1 bar, 298 K}} + \int_{298}^{T} C_P\,dT,$$

$$\mathcal{S}_{\text{1 bar, T}} = \mathcal{S}_{\text{1 bar, 298 K}} + \int_{298}^{T} \frac{C_P}{T}\,dT,$$

$$\int_{\text{1 bar}}^{P} V(P,T)\,dP = PV_0\left(1 - a + \left(a\frac{(1-bP_{th})^{1-c} - (1+b(P-P_{th}))^{1-c}}{b(c-1)P}\right)\right)$$

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 thermal pressure is calculated using a Mie-Grüneisen-like function:

$$P_{\text{th}} = \frac{\alpha_0 K_0 E_{\text{th}}}{C_{V0}},$$
$$E_{\text{th}} = 3nR\Theta \left(0.5 + \frac{1}{\exp(\frac{\Theta}{T}) - 1} \right),$$
$$C_V = 3nR \frac{(\frac{\Theta}{T})^2 \exp(\frac{\Theta}{T})}{(\exp(\frac{\Theta}{T}) - 1)^2}$$

Θ is the Einstein temperature of the crystal in Kelvin, approximated for a substance i with n_i atoms in the unit formula and a molar entropy S_i using the empirical formula:

$$\Theta_i = \frac{10636}{S_i/n_i + 6.44}$$

Parameter	Description	Units
`P_0`	Reference pressure.	Pa
`T_0`	Reference temperature.	K
`H_0`	Enthalpy at the reference state.	J/mol
`S_0`	Entropy at the reference state.	J/(mol K)
`Cp`	Heat capacity coefficients at the reference pressure (length 4 list).	Various
`V_0`	Volume at the reference state.	m^3/mol
`K_0`	Isothermal bulk modulus at the reference state.	Pa
`Kprime_0`	Pressure derivative of bulk modulus at the reference state.	Dimensionless
`Kdprime_0`	Second pressure derivative of bulk modulus at the reference state.	Dimensionless
`a_0`	Thermal expansivity at reference state.	1/K
`T_einstein`	Einstein temperature.	K
`n`	Number of atoms in the unit formula.	Dimensionless

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]

isothermal_bulk_modulus_reuss(*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.

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]. This function replaces -Pth in Equation 13+1 in [HollandPowell11] with P-Pth for non-ambient temperature

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

class burnman.eos.hp.**HP_TMTL**

Bases: *EquationOfState*

Base class for the thermal equation of state described in [HollandPowell98], but with the Modified Tait as the static part, as described in [HollandPowell11].

Parameter	Description	Units
P_0	Reference pressure.	Pa
T_0	Reference temperature.	K
H_0	Enthalpy at the reference state.	J/mol
S_0	Entropy at the reference state.	J/(mol K)
Cp	Heat capacity coefficients at the reference pressure (length 4 list).	Various
V_0	Volume at the reference state.	m^3/mol
K_0	Isothermal bulk modulus at the reference state.	Pa
Kprime_0	Pressure derivative of bulk modulus at the reference state.	Dimensionless
Kdprime_0	Second pressure derivative of bulk modulus at the reference state.	Dimensionless
a_0	Thermal expansivity at reference state.	1/K
dKdT_0	Temperature derivative of bulk modulus at reference state.	Pa/K

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

isothermal_bulk_modulus_reuss(*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.

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity at the pressure, temperature, and volume [1/K]

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the gibbs free energy [J/mol] as a function of pressure [Pa] and temperature [K].

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy [J/K/mol] as a function of pressure [Pa] and temperature [K].

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns the heat capacity [J/K/mol] as a function of pressure [Pa] and temperature [K].

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

pressure(*temperature*, *volume*, *params*)

Parameters

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Pressure of the mineral, including cold and thermal parts $[m^3]$.

Return type
float

20.9.2.8 mie grueneisen debye

class burnman.eos.mie_grueneisen_debye.**MGD2**

Bases: *MGDBase*

MGD equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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

class burnman.eos.mie_grueneisen_debye.**MGD3**

Bases: *MGDBase*

Mie-Grueneisen-Debye equation of state with third order finite strain expansion for the shear modulus.

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 [SLB05]. 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.

Parameter	Description	Units
F_0	Reference Helmholtz free energy.	J/mol
V_0	Reference volume.	m^3
K_0	Reference bulk modulus.	Pa
Kprime_0	Pressure derivative of the bulk modulus at reference pressure.	Dimensionless
molar_mass	Molar mass of the mineral.	kg/mol
n	Number of atoms per formula unit.	Dimensionless
Debye_0	Debye temperature at reference state.	K
grueneisen_0	Grüneisen parameter at reference state.	Dimensionless
q_0	Volume dependence of the Grüneisen parameter.	Dimensionless

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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

class burnman.eos.mie_grueneisen_debye.**MGDBase**

Bases: *EquationOfState*

Base class for a generic finite-strain Mie-Grueneisen-Debye equation of state. References for this can be found in many places, such as [SDT02] and [JR96]. Here we mostly follow the appendices of [MBR+07]. Of particular note is the thermal correction to the shear modulus, which was developed by [HS98].

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] EQ B7

isothermal_bulk_modulus_reuss(*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

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]

pressure(*temperature*, *volume*, *params*)

Returns pressure [Pa] as a function of temperature [K] and volume[m^3] EQ B7

gibbs_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]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.2.9 modular mie grueneisen debye

class burnman.eos.modular_mie_grueneisen_debye.**ModularMGD**

Bases: *EquationOfState*

This class implements a modular Mie-Grueneisen-Debye (MGD) equation of state (EoS) where the reference (isothermal) EoS and Debye temperature model can be chosen by the user.

The equation of state also supports electronic contributions to the Helmholtz energy and thermodynamic properties according to the model of [BK77].

Parameter	Description	Units
V_0	Reference volume	m^3
T_0	Reference temperature	K
bel_0	Electronic contribution parameter	unitless
gel	Electronic contribution exponent	unitless
n	Number of atoms per formula unit	atoms per formula unit
molar_mass	Molar mass	kg/mol
reference_eos	Reference isothermal equation of state. Must have methods for pressure, isothermal bulk modulus, and Gibbs energy.	*burnman.eos.EquationOfState*
debye_temperatu	Debye temperature model. An object providing the Debye temperature as a function of relative volume via value, and its first and second derivatives with respect to relative volume via dVrel and dVrel2.	callable. Examples include *burnman.eos.debye_temperature_models.SLB*, *burnman.eos.debye_temperature_models.PowerLawGammaSimple*, and *burnman.eos.debye_temperature_models.PowerLawGamma*.
anharmonic_ther	Anharmonic thermal model. An object providing the anharmonic contribution to the energy.	callable. Examples include *burnman.eos.anharmonic_thermal_models.Pade* and *burnman.eos.anharmonic_thermal_models.LogNormal*.

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

pressure(*temperature*, *volume*, *params*)

Returns the pressure of the mineral at a given temperature and volume [Pa]

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Shear modulus of the mineral. $[Pa]$

Return type
float

class burnman.eos.modular_mie_grueneisen_debye.**ModularMGDWithAnharmonicity**

Bases: *ModularMGD*

This class extends the ModularMGD class to include anharmonicity effects according to a simplification of the model proposed by Wu and Wentzcovitch, 2009.

The basis of the anharmonic model is the definition of a scaled volume, V': $\ln(V'/V) == -c\ln(V/V_0)$. In this expression, V is the target volume, V_0 is a first order approximation to the volume at the same pressure and reference temperature T_0, and c is an anharmonicity parameter provided in the params dictionary as *c_anh*.

The anharmonic Helmholtz energy F is related to the scaled volume by the equation:

$$F(V,T) = F_h(V',T) + F_h(V,T_0) - F_h(V',T_0)$$

where F_h is the harmonic Helmholtz energy, potentially with electronic contributions.

Note: This model is not the same as that published in Wu and Wentzcovitch (2009). The results are expected to be similar, but the c parameter will in general need to be changed. This is because only a local approximation to the volume change between 0 K and the target temperature is used. This does not mean that the model is less able to capture the essential physics of the problem; indeed, the model of Wu and Wentzcovitch (2009) is only intended to be an effective ansatz.

Parameter	Description	Units
V_0	Reference volume	m^3
T_0	Reference temperature	K
bel_0	Electronic contribution parameter	unitless
gel	Electronic contribution exponent	unitless
n	Number of atoms per formula unit	atoms per formula unit
molar_mass	Molar mass	kg/mol
reference_eos	Reference isothermal equation of state. Must have methods for pressure, isothermal bulk modulus, and Gibbs energy.	*burnman.eos.EquationOfState*
debye_temperatu	Debye temperature model. An object providing the Debye temperature as a function of relative volume via value, and its first and second derivatives with respect to relative volume via dVrel and dVrel2.	callable. Examples include *burnman.eos.debye_temperature_models.SLB*, *burnman.eos.debye_temperature_models.PowerLawGammaSimple*, and *burnman.eos.debye_temperature_models.PowerLawGamma*.
anharmonic_ther	Anharmonic thermal model. An object providing the anharmonic contribution to the energy.	callable. Examples include *burnman.eos.anharmonic_thermal_models.Pade* and *burnman.eos.anharmonic_thermal_models.LogNormal*.
c_anh	Anharmonicity parameter	unitless

lnVoverV0_approx(*temperature*, *volume*, *params*)

pressure()

(copied from pressure):

Returns the pressure of the mineral at a given temperature and volume [Pa]

isothermal_bulk_modulus_reuss()

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Shear modulus of the mineral. $[Pa]$

Return type
float

volume(*pressure*, *temperature*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K]

20.9.2.10 slb

class burnman.eos.slb.**SLB2**

Bases: *SLBBase*

SLB equation of state with second order finite strain expansion for the shear modulus. In general, this should not be used, but sometimes shear modulus data is fit to a second order equation of state. In that case, you should use this. The moral is, be careful!

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Density of the mineral. $[kg/m^3]$

Return type
float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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. $[m^3]$

class burnman.eos.slb.**SLB3**

Bases: *SLBBase*

Third-order finite strain-Mie-Grueneisen-Debye equation of state detailed in [SLB05].

This equation of state combines the third-order Birch-Murnaghan isothermal equation of state with thermal corrections derived from the Mie-Grüneisen-Debye equation of state with the quasi-harmonic approximation:

$$\mathcal{F}_{th}(T,\theta(V)) = nRT\left(3\ln(1-e^{-\frac{\theta}{T}}) - \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau-1)}d\tau\right),$$

$$P_{th}(V,T) = \frac{\gamma\Delta\mathcal{U}}{V},$$

$$K_{th}(V,T) = (\gamma+1-q)\frac{\gamma\Delta\mathcal{U}}{V} - \gamma^2\frac{\Delta(C_V T)}{V},$$

$$G_{th}(V,T) = -\frac{\eta_S\Delta\mathcal{U}}{V}.$$

The expression for the Debye temperature $\theta(V)$ as a function of volume is given by:

$$\theta(V) = \theta_0 \sqrt{\frac{\nu^2}{\nu_0^2}},$$
$$\frac{\nu^2}{\nu_0^2} = 1 + a_{ii}^{(1)} f + \frac{1}{2} a_{iikk}^{(2)} f^2,$$
$$f = \frac{1}{2}\left[\left(\frac{V_0}{V}\right)^{\frac{2}{3}} - 1\right].$$

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:

$$C_V = 9nR\left(\frac{T}{\theta}\right)^3 \int_0^{\frac{\theta}{T}} \frac{e^\tau \tau^4}{(e^\tau - 1)^2} d\tau,$$
$$\mathcal{U} = 9nRT\left(\frac{T}{\theta}\right)^3 \int_0^{\frac{\theta}{T}} \frac{\tau^3}{(e^\tau - 1)} d\tau,$$
$$\gamma = \frac{1}{6}\frac{\nu_0^2}{\nu^2}(2f+1)\left[a_{ii}^{(1)} + a_{iikk}^{(2)} f\right],$$
$$q = \frac{1}{9\gamma}\left[18\gamma^2 - 6\gamma - \frac{1}{2}\frac{\nu_0^2}{\nu^2}(2f+1)^2 a_{iikk}^{(2)}\right],$$
$$\eta_S = -\gamma - \frac{1}{2}\frac{\nu_0^2}{\nu^2}(2f+1)^2 a_S^{(2)},$$
$$a_{ii}^{(1)} = 6\gamma_0,$$
$$a_{iikk}^{(2)} = -12\gamma_0 + 36\gamma_0^2 - 18 q_0 \gamma_0,$$
$$a_S^{(2)} = -2\gamma_0 - 2\eta_{S0},$$

where 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.

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].

Parameter	Description	Units
V_0	Volume at P = 10^5 Pa , T = 300 K	m^3/mol
K_0	Isothermal bulk modulus at P = 10^5 Pa, T = 300 K	Pa
Kprime_0	Pressure derivative of K_0	unitless
G_0	Shear modulus at P = 10^5 Pa, T = 300 K	Pa
Gprime_0	Pressure derivative of G_0	unitless
molar_mass	mass per mole formula unit	kg mol^{-1}
n	number of atoms per formula unit	atoms/formula unit
Debye_0	Debye Temperature	K
grueneisen_0	Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
q_0	Volume dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
eta_s_0	Shear strain dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless

This equation of state is substantially the same as Mie-Gruneisen-Debye. 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.

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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. $[m^3]$

class burnman.eos.slb.**SLB3Conductive**

Bases: *SLBBase*

SLB equation of state with third order finite strain expansion for the shear modulus and a contribution to the Helmholtz free energy that arises from the thermal excitation of electrons [BK77].

For more details on the underlying EoS, we refer readers to [SLB05] and the documentation for *burnman.eos.slb.SLB3*. The electronic contribution to the Helmholtz free energy is given by:

$$\mathcal{F}_{el}(V,T) = -\frac{1}{2} b_0^{el} V \left(\frac{V}{V_0}\right)^{g^{el}} (T^2 - T_0^2)$$

Parameter	Description	Units
V_0	Volume at P = 10^5 Pa , T = 300 K	m^3/mol
K_0	Isothermal bulk modulus at P = 10^5 Pa, T = 300 K	Pa
Kprime_0	Pressure derivative of K_0	unitless
G_0	Shear modulus at P = 10^5 Pa, T = 300 K	Pa
Gprime_0	Pressure derivative of G_0	unitless
molar_mass	mass per mole formula unit	kg mol^{-1}
n	number of atoms per formula unit	atoms/formula unit
Debye_0	Debye Temperature	K
grueneisen_0	Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
q_0	Volume dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
eta_s_0	Shear strain dependence of Grüneisen parameter at P = 10^5 Pa, T = 300 K	unitless
bel_0	Electronic bulk modulus coefficient	Pa
gel	Electronic bulk modulus volume exponent	unitless

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Enthalpy of the mineral $[J/mol]$.

Return type
float

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

gibbs_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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

isothermal_bulk_modulus_reuss(*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*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

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. $[m^3]$

class burnman.eos.slb.**SLBBase**

Bases: *EquationOfState*

Base class for the finite strain-Mie-Grueneiesen-Debye equation of state detailed in [SLB05]. For the most part the equations are all third order in strain, but see further the burnman.slb.SLB2 and burnman.slb.SLB3 classes.

volume(*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]

isothermal_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Returns isothermal bulk modulus $[Pa]$

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns shear modulus. $[Pa]$

thermal_expansivity(*pressure*, *temperature*, *volume*, *params*)

Returns thermal expansivity. $[1/K]$

entropy(*pressure*, *temperature*, *volume*, *params*)

Returns the entropy at the pressure and temperature of the mineral [J/K/mol]

molar_heat_capacity_p(*pressure*, *temperature*, *volume*, *params*)

Returns heat capacity at constant pressure. $[J/K/mol]$

gibbs_energy(*pressure*, *temperature*, *volume*, *params*)

Returns the Gibbs free energy at the pressure and temperature of the mineral [J/mol]

validate_parameters(*params*)

Check for existence and validity of the parameters

density(*volume*, *params*)

Calculate the density of the mineral $[kg/m^3]$. The params object must include a "molar_mass" field.

Parameters

- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Density of the mineral. $[kg/m^3]$

Return type

float

enthalpy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns

Enthalpy of the mineral $[J/mol]$.

Return type

float

grueneisen_parameter(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.

- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Grueneisen parameter of the mineral. $[unitless]$

Return type
float

helmholtz_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Helmholtz energy of the mineral $[J/mol]$.

Return type
float

isentropic_bulk_modulus_reuss(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Adiabatic bulk modulus of the mineral. $[Pa]$

Return type
float

molar_heat_capacity_v(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.

- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Heat capacity at constant volume of the mineral. $[J/K/mol]$

Return type
float

molar_internal_energy(*pressure*, *temperature*, *volume*, *params*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the equation of state $[Pa]$.
- **temperature** (*float*) – Temperature at which to evaluate the equation of state $[K]$.
- **volume** (*float*) – Molar volume of the mineral. For consistency this should be calculated using *volume()* $[m^3]$.
- **params** (*dict*) – Dictionary containing material parameters required by the equation of state.

Returns
Internal energy of the mineral $[J/mol]$.

Return type
float

20.9.3 Other Equation of State classes

20.9.3.1 anharmonic debye

class burnman.eos.anharmonic_debye.**AnharmonicDebye**

Bases: object

Class providing methods to compute the anharmonic contribution to the Helmholtz free energy, pressure, entropy, isochoric heat capacity, isothermal bulk modulus and thermal expansion coefficient multiplied by the isothermal bulk modulus.

The Helmholtz energy is defined as $F(V,T) = A(V)(F_a(T,\Theta(V)) - F_a(T_0,\Theta(V)))$ where $A(V)$ is the anharmonic prefactor, $F_a(T,\Theta)$ is the anharmonic Helmholtz energy, and $\Theta(V)$ is the Debye temperature.

These three functions and their derivatives with respect to their arguments are provided as class instances contained within the params dictionary, with keys "anharmonic_prefactor_model", "debye_temperature_model", and "anharmonic_thermal_model".

Returns
description

Return type
type

static helmholtz_energy(*temperature*, *volume*, *params*)

static entropy(*temperature*, *volume*, *params*)

static heat_capacity_v(*temperature*, *volume*, *params*)

static pressure(*temperature*, *volume*, *params*)

static isothermal_bulk_modulus(*temperature*, *volume*, *params*)

static dSdV(*temperature*, *volume*, *params*)

static validate_parameters(*params*)

20.9.3.2 anharmonic prefactor models

class burnman.eos.anharmonic_prefactor_models.**AnharmonicPrefactorModel**

Bases: object

Base class for anharmonic prefactor models.

value(*Vrel*, *params*)

dVrel(*Vrel*, *params*)

dVrel2(*Vrel*, *params*)

validate_parameters(*params*)

class burnman.eos.anharmonic_prefactor_models.**PowerLaw**

Bases: *AnharmonicPrefactorModel*

Class providing methods to compute the prefactor A in the anharmonic contribution to thermodynamic models.

The prefactor is defined as $A = a_{anh} * (V/V_0)^{m_{anh}}$, with both a_{anh} and m_{anh} being parameters of the model.

Returns
description

Return type
type

value(*Vrel*, *params*)

dVrel(*Vrel*, *params*)

dVrel2(*Vrel*, *params*)

validate_parameters(*params*)

class burnman.eos.anharmonic_prefactor_models.**Sigmoid**

Bases: *AnharmonicPrefactorModel*

Class providing methods to compute the prefactor A in the anharmonic contribution to thermodynamic models.

The prefactor is defined as $A = a_{anh} * (1 - 1/(1 + (Vrel + b_{anh}) * *c_{anh}))$ where a_{anh}, b_{anh} and c_{anh} are parameters of the model.

Returns
description

Return type
type

value(*Vrel*, *params*)

dVrel(*Vrel*, *params*)

dVrel2(*Vrel*, *params*)

validate_parameters(*params*)

20.9.3.3 anharmonic thermal models

class burnman.eos.anharmonic_thermal_models.**AnharmonicThermalModelBase**

Bases: object

Abstract base class for anharmonic thermal models.

nondimensional_helmholtz_energy(*T*, *debye_T*, *params*)

Anharmonic contribution to the Helmholtz free energy at a given temperature.

nondimensional_entropy(*T*, *debye_T*, *params*)

Anharmonic contribution to the entropy at a given temperature.

nondimensional_heat_capacity(*T*, *debye_T*, *params*)

Anharmonic contribution to the heat capacity at a given temperature.

nondimensional_dhelmholtz_dTheta(*T*, *debye_T*, *params*)

Anharmonic contribution to the derivative of the Helmholtz energy with respect to the Debye temperature.

nondimensional_d2helmholtz_dTheta2(*T*, *debye_T*, *params*)

Anharmonic contribution to the second derivative of the Helmholtz energy with respect to the Debye temperature.

nondimensional_dentropy_dTheta(*T*, *debye_T*, *params*)

Anharmonic contribution to the derivative of the entropy with respect to the Debye temperature.

20.9.3.4 debye temperature models

class burnman.eos.debye_temperature_models.**DebyeTemperatureModelBase**

Bases: ABC

Abstract base class for Debye temperature models.

abstractmethod value(*Vrel: float*, *params: dict*) → float

Return the Debye temperature for the given relative volume.

dVrel(*Vrel: float*, *params: dict*) → float

First derivative of Debye temperature wrt relative volume.

dVrel2(*Vrel: float*, *params: dict*) → float

Second derivative of Debye temperature wrt relative volume.

20.9.4 Equation of State Functions

20.9.4.1 anharmonic thermal models

burnman.eos.anharmonic_thermal_models.**LogNormal**(**args*, ***kwargs*)

burnman.eos.anharmonic_thermal_models.**Pade**(**args*, ***kwargs*)

20.9.4.2 birch murnaghan

burnman.eos.birch_murnaghan.**bulk_modulus_fourth_order**(*volume*, *params*)

Bulk modulus for the fourth order Birch-Murnaghan equation of state.

Parameters

- **volume** (*float*) – Volume of the material in the same units as the reference volume.
- **params** (*dict*) – Parameter dictionary

Returns

Bulk modulus in the same units as the reference bulk modulus.

Return type

float

burnman.eos.birch_murnaghan.**bulk_modulus_third_order**(*volume*, *params*)

Bulk modulus for the third order Birch-Murnaghan equation of state [Bir47].

Parameters

- **volume** (*float*) – Volume of the material in the same units as the reference volume.
- **params** (*dict*) – Parameter dictionary

Returns

Bulk modulus in the same units as the reference bulk modulus.

Return type
float

burnman.eos.birch_murnaghan.**pressure_fourth_order**(*invVrel*, *params*)

Pressure for the fourth order Birch-Murnaghan equation of state.

Parameters

- **invVrel** (*float*) – Reference volume divided by the volume
- **params** (*dict*) – Parameter dictionary

Returns
Pressure in the same units that are supplied for the reference bulk modulus (params['K_0']).

Return type
float

burnman.eos.birch_murnaghan.**pressure_third_order**(*invVrel*, *params*)

Pressure for the third order Birch-Murnaghan equation of state.

Parameters

- **invVrel** (*float*) – Reference volume divided by the volume
- **params** (*dict*) – Parameter dictionary

Returns
Pressure in the same units that are supplied for the reference bulk modulus (params['K_0']).

Return type
float

burnman.eos.birch_murnaghan.**shear_modulus_second_order**(*volume*, *params*)

Shear modulus for the second order Birch Murnaghan equation of state (i.e. expanded to 2nd order in strain).

Parameters

- **volume** (*float*) – Molar volume in the same units as the reference volume.
- **params** (*dict*) – Parameter dictionary

Returns
Shear modulus in the same units as the reference shear modulus.

Return type
float

burnman.eos.birch_murnaghan.**shear_modulus_third_order**(*volume*, *params*)

Shear modulus for the third order Birch Murnaghan equation of state.

Parameters

- **volume** (*float*) – Molar volume in the same units as the reference volume.

- **params** (*dict*) – Parameter dictionary

Returns
Shear modulus in the same units as the reference shear modulus.

Return type
float

burnman.eos.birch_murnaghan.**volume_fourth_order**(*pressure*, *params*)

Volume for the fourth order Birch-Murnaghan equation of state.

Parameters

- **pressure** (*float*) – Pressure in the same units that are supplied for the reference bulk modulus (params['K_0']).
- **params** (*dict*) – Parameter dictionary

Returns
Molar volume in the same units as the reference volume.

Return type
float

burnman.eos.birch_murnaghan.**volume_third_order**(*pressure*, *params*)

Volume for the third order Birch-Murnaghan equation of state.

Parameters

- **pressure** (*float*) – Pressure in the same units that are supplied for the reference bulk modulus (params['K_0']).
- **params** (*dict*) – Parameter dictionary

Returns
Molar volume in the same units as the reference volume.

Return type
float

20.9.4.3 bukowinski electronic

burnman.eos.bukowinski_electronic.**CVoverT**(*volume*, *V_0*, *bel_0*, *gel*)

CV = T dS/dT

burnman.eos.bukowinski_electronic.**KToverV**(*temperature*, *volume*, *T_0*, *V_0*, *bel_0*, *gel*)

KT = -V dP/dV

burnman.eos.bukowinski_electronic.**aKT**(*temperature*, *volume*, *V_0*, *bel_0*, *gel*)

aKT = dP/dT

burnman.eos.bukowinski_electronic.**entropy**(*temperature*, *volume*, *V_0*, *bel_0*, *gel*)

S = -dF/dT

burnman.eos.bukowinski_electronic.**helmholtz**(*temperature*, *volume*, *T_0*, *V_0*, *bel_0*, *gel*)

burnman.eos.bukowinski_electronic.**pressure**(*temperature*, *volume*, *T_0*, *V_0*, *bel_0*, *gel*)

P = -dF/dV

20.9.4.4 debye

burnman.eos.debye.**d2helmholtz_dTheta2**(*T*, *debye_T*, *n*)

Second derivative of the Helmholtz free energy with respect to the Debye temperature [J/K^2].

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.**dentropy_dTheta**(*T*, *debye_T*, *n*)

First derivative of the entropy with respect to the Debye temperature [J/K].

burnman.eos.debye.**dhelmholtz_dTheta**(*T*, *debye_T*, *n*)

First derivative of the Helmholtz free energy with respect to the Debye temperature [J/K].

burnman.eos.debye.**dmolar_heat_capacity_v_dT**(*T*, *debye_T*, *n*)

First temperature derivative of the heat capacity at constant volume [J/K^2/mol].

burnman.eos.debye.**entropy**(*T*, *debye_T*, *n*)

Entropy due to lattice vibrations in the Debye model [J/K].

burnman.eos.debye.**helmholtz_energy**(*T*, *debye_T*, *n*)

Helmholtz free energy of lattice vibrations in the Debye model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyways.

burnman.eos.debye.**molar_heat_capacity_v**(*T*, *debye_T*, *n*)

Heat capacity at constant volume. In J/K/mol

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

20.9.4.5 debye temperature models

burnman.eos.debye_temperature_models.**PowerLawGamma**(**args*, ***kwargs*)

burnman.eos.debye_temperature_models.**PowerLawGammaSimple**(**args*, ***kwargs*)

burnman.eos.debye_temperature_models.**SLB**(**args*, ***kwargs*)

20.9.4.6 einstein

burnman.eos.einstein.**d2helmholtz_dTheta2**(*T*, *einstein_T*, *n*)

Second derivative of Helmholtz free energy with respect to Einstein temperature [J/K^2]

burnman.eos.einstein.**dentropy_dTheta**(*T*, *einstein_T*, *n*)

Derivative of Einstein entropy with respect to Einstein temperature [J/K^2]

burnman.eos.einstein.**dhelmholtz_dTheta**(*T*, *einstein_T*, *n*)

First derivative of Helmholtz free energy with respect to Einstein temperature [J/K]

burnman.eos.einstein.**dmolar_heat_capacity_v_dT**(*T*, *einstein_T*, *n*)

First temperature derivative of the heat capacity at constant volume according to the Einstein model [J/K^2/mol].

burnman.eos.einstein.**entropy**(*T*, *einstein_T*, *n*)

Entropy due to lattice vibrations in the Einstein model [J/K]

burnman.eos.einstein.**helmholtz_energy**(*T*, *einstein_T*, *n*)

Helmholtz free energy of lattice vibrations in the Einstein model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyway.

burnman.eos.einstein.**molar_heat_capacity_v**(*T*, *einstein_T*, *n*)

Heat capacity at constant volume. In J/K/mol

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

20.9.4.7 macaw

burnman.eos.macaw.**make_params**(*K0*, *K0_prime*, *K_infinity_prime*)

20.9.4.8 modified tait

burnman.eos.modified_tait.**bulk_modulus**(*pressure*, *params*)

Returns isothermal bulk modulus K_T of the mineral. $[Pa]$. EQ 13+2

burnman.eos.modified_tait.**intVdP**(*pressure*, *params*)

Returns the integral of VdP for the mineral. $[J]$. EQ 13

burnman.eos.modified_tait.**pressure_modified_tait**(*Vrel*, *params*)

Pressure according to the modified Tait equation of state. Equation 2 in [HollandPowell11].

Parameters

- **Vrel** (*float or numpy array*) – Volume divided by the reference volume.
- **params** (*dictionary*) – Parameter dictionary

Returns

pressure in the same units that are supplied for the reference bulk modulus (params['K_0'])

Return type

float or numpy array

burnman.eos.modified_tait.**tait_constants**(*params*)

returns parameters for the modified Tait equation of state derived from K_T and its two first pressure derivatives EQ 4 from Holland and Powell, 2011

burnman.eos.modified_tait.**volume**(*pressure*, *params*)

Returns volume [m^3] as a function of pressure [Pa] and temperature [K] Equation 12 in [Holland-Powell11].

20.9.4.9 morse potential

burnman.eos.morse_potential.**bulk_modulus**(*volume*, *params*)

Compute the bulk modulus as per the Morse potential equation of state [Mor29]. Returns bulk modulus in the same units as the reference bulk modulus. Pressure must be in $[Pa]$.

burnman.eos.morse_potential.**morse_potential**(*VoverV0*, *params*)

Equation for the Morse Potential equation of state, returns pressure in the same units that are supplied for the reference bulk modulus (params['K_0'])

burnman.eos.morse_potential.**shear_modulus**(*volume*, *params*)

Shear modulus not currently implemented for this equation of state

burnman.eos.morse_potential.**volume**(*pressure*, *params*)

Get the Morse Potential volume at a reference temperature for a given pressure $[Pa]$. Returns molar volume in $[m^3]$

20.9.4.10 murnaghan

burnman.eos.murnaghan.**bulk_modulus**(*pressure*, *K_0*, *Kprime_0*)

burnman.eos.murnaghan.**energy**(*volume*, *E_0*, *V_0*, *K_0*, *Kprime_0*)

burnman.eos.murnaghan.**intVdP**(*pressure*, *V_0*, *K_0*, *Kprime_0*)

burnman.eos.murnaghan.**pressure**(*volume*, *V_0*, *K_0*, *Kprime_0*)

burnman.eos.murnaghan.**volume**(*pressure*, *V_0*, *K_0*, *Kprime_0*)

20.9.4.11 property modifiers

burnman.eos.property_modifiers.**bragg_williams_excesses**(*pressure*, *temperature*, *params*)

The Bragg-Williams model is a symmetric solution model with an excess configurational entropy term determined by the specifics of order-disorder in the mineral, multiplied by some empirical factor. Expressions for the excess Gibbs free energy can be found in [HP96].

Excess properties assume that order-disorder processes are rapid compared with the timescales of pressure and temperature changes (i.e., equilibrium is maintained).

This may not be reasonable for order-disorder, especially for slow or coupled diffusers (Si-Al, for example). Properties for minerals that order-disorder slowly should be calculated using an explicit solid solution model.

Table 1: Parameters

Parameter	Description
`deltaH`	Enthalpy change (J/mol)
`deltaV`	Volume change (m^3/mol)
`Wh`	Enthalpy interaction parameter (J/mol)
`Wv`	Volume interaction parameter (m^3/mol)
`n`	Related to the number of available sites for ordering
`factor`	An empirical factor

burnman.eos.property_modifiers.**calculate_property_modifications**(*mineral*)

Sums the excesses from all the modifiers.

To calculate thermodynamic properties from the outputs, the following functions should be used (the _o suffix stands for original value):

gibbs = gibbs_o + excesses['G'] S = S_o - excesses['dGdT'] V = V_o + excesses['dGdP'] K_T = V / ((V_o / K_T_o) - excesses['d2GdP2']) C_p = C_p_o - temperature*excesses['d2GdT2'] alpha = ((alpha_o*V_o) + excesses['d2GdPdT']) / V

H = gibbs + temperature*S helmholtz = gibbs - pressure*V C_v = C_p - V*temperature*alpha*alpha*K_T gr = alpha*K_T*V/C_v K_S = K_T*C_p/C_v

burnman.eos.property_modifiers.**debye_delta_excesses**(*pressure*, *temperature*, *params*)

Applies an excess contribution based on a Debye model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= -U_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -C_{V,\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{dC_{V,\text{Debye}}}{dT}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess entropy tends toward a constant value at high temperature and behaves like the heat capacity of a Debye model at finite temperature.

Table 2: Parameters

Parameter	Description
S_inf	Entropy at infinite temperature (J/mol/K).
Theta_0	Einstein temperature (K).

burnman.eos.property_modifiers.**debye_excesses**(*pressure*, *temperature*, *params*)

Applies an excess contribution based on a Debye model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= F_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -S_{\text{Debye}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{C_{V,\text{Debye}}}{T}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess heat capacity tends toward a constant value at high temperature.

Table 3: Parameters

Parameter	Description
Cv_inf	Heat capacity at infinite temperature (J/mol/K).
Theta_0	Debye temperature (K).

burnman.eos.property_modifiers.**einstein_delta_excesses**(*pressure*, *temperature*, *params*)

Applies an excess contribution based an Einstein model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= -U_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -C_{V,\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{dC_{V,\text{Einstein}}}{dT}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess entropy tends toward a constant value at high temperature and behaves like the heat capacity of an Einstein model at finite temperature.

Table 4: Parameters

Parameter	Description
S_inf	Entropy at infinite temperature (J/mol/K).
Theta_0	Einstein temperature (K).

burnman.eos.property_modifiers.**einstein_excesses**(*pressure*, *temperature*, *params*)

Applies an excess contribution based an Einstein model. The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= F_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial T} &= -S_{\text{Einstein}}, \\
\frac{\partial \mathcal{G}}{\partial P} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= -\frac{C_{V,\text{Einstein}}}{T}, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

The excess heat capacity tends toward a constant value at high temperature.

Table 5: Parameters

Parameter	Description
Cv_inf	Heat capacity at infinite temperature (J/mol/K).
Theta_0	Einstein temperature (K).

burnman.eos.property_modifiers.**landau_excesses**(*pressure*, *temperature*, *params*)

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.

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:

$$\mathcal{G}_{\text{dis}} = -S_D \left((T - Tc) + \frac{Tc_0}{3} \right),$$
$$\frac{\partial \mathcal{G}}{\partial P}_{\text{dis}} = V_D,$$
$$\frac{\partial \mathcal{G}}{\partial T}_{\text{dis}} = -S_D$$

If temperature is below the critical temperature, Q is between 0 and 1. The Gibbs energy can be described thus:

$$Q^2 = \sqrt{\left(1 - \frac{T}{Tc}\right)},$$
$$\mathcal{G} = S_D \left((T - Tc)Q^2 + \frac{Tc_0 Q^6}{3} \right) + \mathcal{G}_{\text{dis}},$$
$$\frac{\partial \mathcal{G}}{\partial P} = -V_D Q^2 \left(1 + \frac{T}{2Tc} \left(1. - \frac{Tc_0}{Tc} \right) \right) + \frac{\partial \mathcal{G}}{\partial P}_{\text{dis}},$$
$$\frac{\partial \mathcal{G}}{\partial T} = S_D Q^2 \left(\frac{3}{2} - \frac{Tc_0}{2Tc} \right) + \frac{\partial \mathcal{G}}{\partial T}_{\text{dis}},$$
$$\frac{\partial^2 \mathcal{G}}{\partial P^2} = V_D^2 \frac{T}{S_D Tc^2 Q^2} \left(\frac{T}{4Tc} \left(1. + \frac{Tc_0}{Tc} \right) + Q^4 \left(1. - \frac{Tc_0}{Tc} \right) - 1 \right),$$
$$\frac{\partial^2 \mathcal{G}}{\partial T^2} = -\frac{S_D}{TcQ^2} \left(\frac{3}{4} - \frac{Tc_0}{4Tc} \right),$$
$$\frac{\partial^2 \mathcal{G}}{\partial P \partial T} = \frac{V_D}{2TcQ^2} \left(1 + \left(\frac{T}{2Tc} - Q^4 \right) \left(1 - \frac{Tc_0}{Tc} \right) \right)$$

Table 6: Parameters

Parameter	Description
`Tc_0`	Critical temperature at reference pressure (K)
`S_D`	Entropy parameter (J/mol/K)
`V_D`	Volume parameter (m^3/mol)

`burnman.eos.property_modifiers.`**`landau_hp_excesses`**(*pressure*, *temperature*, *params*)

Applies a tricritical Landau correction similar to that described above. However, this implementation follows [HollandPowell11], who compute properties relative to the standard state.

Includes the correction published within landaunote.pdf (Holland, pers. comm), which 'corrects' the terms involving the critical temperature Tc / Tc*.

Note that this implementation allows the order parameter Q to be greater than one.

$$Tc = Tc0 + \frac{V_D P}{S_D}$$

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

$$\frac{\partial^2 \mathcal{G}}{\partial P^2} = -\frac{V_D^2}{2S_D Tc_0 Q^2},$$
$$\frac{\partial^2 \mathcal{G}}{\partial T^2} = -\frac{S_D}{2Tc_0 Q^2},$$
$$\frac{\partial^2 \mathcal{G}}{\partial P \partial T} = \frac{V_D}{2Tc_0 Q^2}$$

Table 7: Parameters

Parameter	Description
`P_0`	Reference pressure (Pa)
`T_0`	Reference temperature (K)
`Tc_0`	Critical temperature at reference pressure (K)
`S_D`	Entropy parameter (J/mol/K)
`V_D`	Volume parameter (m^3/mol)

`burnman.eos.property_modifiers.`**`landau_slb_2022_excesses`**(*pressure*, *temperature*, *params*)

Applies a tricritical Landau correction to the properties of an endmember which undergoes a displacive phase transition. This correction follows [SLB22], and is done relative to the state with order parameter $Q = 1$.

The order parameter of this formulation can exceed one, at odds with [Putnis92], but in better agreement with atomic intuition [SLB22]. Nevertheless, this implementation is still not perfect, as the excess entropy (and heat capacity) terms are not equal to zero at 0 K. Q is limited to values less than or equal to 2 to avoid unrealistic stabilisation at ultrahigh pressure.

Table 8: Parameters

Parameter	Description
`Tc_0`	Critical temperature at reference pressure (K)
`S_D`	Entropy parameter (J/mol/K)
`V_D`	Volume parameter (m^3/mol)

burnman.eos.property_modifiers.**linear_excesses**(*pressure*, *temperature*, *params*)

A simple linear correction in pressure and temperature.

The excess Gibbs energy and its derivatives are given by:

$$\begin{aligned}
\mathcal{G} &= \Delta\mathcal{E} - T\Delta S + P\Delta V, \\
\frac{\partial \mathcal{G}}{\partial T} &= -\Delta S, \\
\frac{\partial \mathcal{G}}{\partial P} &= \Delta V, \\
\frac{\partial^2 \mathcal{G}}{\partial T^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial P^2} &= 0, \\
\frac{\partial^2 \mathcal{G}}{\partial T \partial P} &= 0
\end{aligned}$$

This form of excess is extremely useful as a first order tweak to free energies (especially in solid solutions where data on all endmembers may not be available).

Table 9: Parameters

Parameter	Description
delta_E	Energy correction (J/mol)
delta_S	Entropy correction (J/mol/K)
delta_V	Volume correction (m^3/mol)

burnman.eos.property_modifiers.**magnetic_excesses_chs**(*pressure*, *temperature*, *params*)

This model approximates the excess energy due to magnetic ordering. It was originally described in [CHS87]. The expressions used in this implementation can be found in [Sun91].

Table 10: Parameters

Parameter	Description
structural_paramet	A dimensionless parameter related to the crystal structure
curie_temperature	A list of length 2: zero pressure Curie temperature (K) and pressure dependence (K/Pa)
magnetic_moment	A list of length 2: zero pressure magnetic moment (Bohr magnetons) and pressure dependence (Bohr magnetons/Pa)

20.9.4.12 reciprocal kprime

burnman.eos.reciprocal_kprime.**bulk_modulus**(*pressure*, *params*)

Returns the bulk modulus at a given pressure

burnman.eos.reciprocal_kprime.**shear_modulus**(*pressure*, *params*)

Returns the shear modulus at a given pressure

20.9.4.13 vinet

burnman.eos.vinet.**bulk_modulus_vinet**(*volume*, *params*)

compute the bulk modulus as per the Vinet equation of state. Reference bulk modulus should be in $[Pa]$.

burnman.eos.vinet.**pressure_vinet**(*V_rel*, *params*)

Pressure equation for the Vinet equation of state, returns pressure in the same units that are supplied for the reference bulk modulus (params['K_0']), which should be in math:*[Pa]*.

burnman.eos.vinet.**volume_vinet**(*pressure*, *params*)

Get the volume at a reference temperature for a given pressure $[Pa]$. Returns molar volume in $[m^3]$

20.10 Solution models

Solution objects in BurnMan are instances of one of three classes: type *Solution* (alias *SolidSolution*), type *ElasticSolution* and type *AnisotropicSolution*.

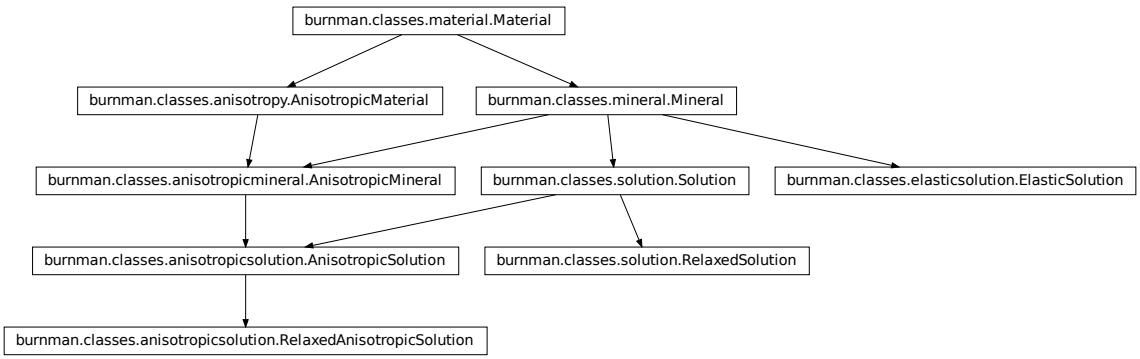
The *Solution* class implements commonly used models (in petrology). Excess properties are defined relative to the endmember properties at fixed pressure and temperature. The formulations are defined with interaction parameters such as excess energies, volumes and entropies.

The *AnisotropicSolution* class extends the *Solution* class to include anisotropic elastic properties.

The standard and anisotropic solution classes can also be "relaxed" via two derived classes: *RelaxedSolution* and *RelaxedAnisotropicSolution*. In these classes, specific linear combinations of molar fractions can be designated as relaxation vectors. These are allowed to vary freely to minimize the Gibbs free energy of the solution.

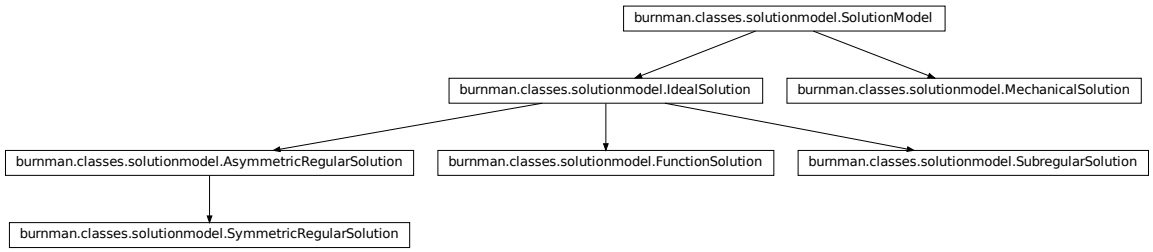
An independent class of solution models is provided by the *ElasticSolution* class. This class instead defines excess properties are relative to the endmember properties at fixed volume and temperature. Such models have their roots in atom-scale considerations; mixing of two instances of the same lattice type requires deformation (local lattice distortions), that can be considered to induce a local chemical stress. Therefore, volume may be a more useful independent variable than pressure. For more details, see [Myh18].

Available solution classes

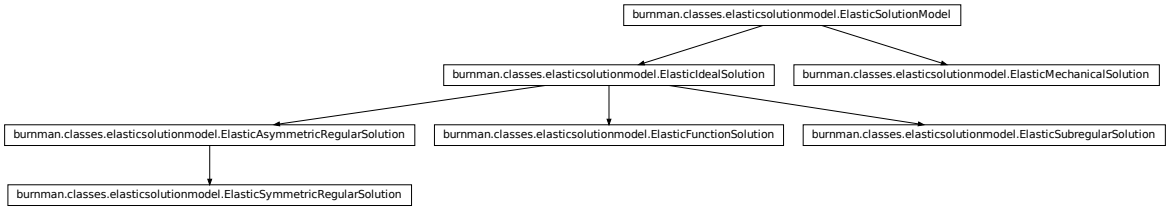


All of the above classes accept several models which define the properties of the solution. The standard and anisotropic solution classes share the same solution models, while the elastic solution class has its own set of closely related solution models.

Available solution model classes



Available elastic solution model classes



20.10.1 Base classes

class burnman.**Solution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

The Solution class has overall responsibility for calculating thermodynamically self-consistent properties of a solution phase.

In this class, solutions are treated as a collection of independent endmember phases (components), plus a model that describes the excess properties of the solution. It stores the current state (pressure, temperature and molar fractions of all the endmembers) of the solution, and interrogates a SolutionModel object to calculate the excess properties of the solution. It then combines these with the properties of the individual endmember phases (also stored in the SolutionModel) to calculate the overall properties

of the solution. This is done using standard thermodynamic relations:

$$G = \sum_i X_i G_i + G^{\text{excess}}$$
$$V = \sum_i X_i V_i + V^{\text{excess}}$$
$$S = \sum_i X_i S_i + S^{\text{excess}}$$
$$C_p = \sum_i X_i C_{p,i} + C_p^{\text{excess}}$$
$$\alpha V = \sum_i X_i \alpha_i V_i + (\alpha V)^{\text{excess}}$$
$$\frac{V}{K_T} = \sum_i X_i \frac{V_i}{K_{T,i}} + \left(\frac{V}{K_T}\right)^{\text{excess}}$$

After instantiation, the user can set the composition of the solution using the set_composition() method, and the pressure and temperature using the set_state() method.

Instantiation of a Solution object can be achieved in just a few lines:

```
from burnman import Solution, SolutionModel

# Create a SolutionModel object (explained in the documentation for that␣
↪class)
solution_model = SolutionModel(...)

# Create a Solution object
solution = Solution(name="MySolution", solution_model=solution_model, molar_
↪fractions=...)
```

where the arguments are as follows:

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.SolutionModel*) – The SolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The initial molar fractions of each end-member in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters
molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution. :rtype: Counter

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies
- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility_reuss

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property isentropic_compressibility_reuss

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns

An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type

numpy.ndarray

property n_reactions

The number of reactions in reaction_basis.

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type

numpy.ndarray

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

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector $b = [b_C, b_O]$, the constraint $1 * b_C - 2 * b_O = 0$ must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property endmember_names

A list of names for all the endmembers in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they

should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

class burnman.**ElasticSolution**(*name=None*, *solution_model=None*, *molar_fractions=None*)

Bases: *Mineral*

This is the base class for all Elastic solutions. Site occupancies, endmember activities and the constant and volume and temperature dependencies of the excess properties can be queried after using set_composition(). States of the solution can only be queried after setting the pressure, temperature and composition using set_state() and set_composition.

This class is available as *burnman.ElasticSolution*. It uses an instance of *burnman.ElasticSolutionModel* to calculate interaction terms between endmembers.

All the solution parameters are expected to be in SI units. This means that the interaction parameters should be in J/mol, with the T and V derivatives in J/K/mol and Pa/mol.

The parameters are relevant to all Elastic solution models. Please see the documentation for individual models for details about other parameters.

Parameters

- **name** (*string*) – Name of the solution.
- **solution_model** (*burnman.ElasticSolutionModel*) – The ElasticSolutionModel object defining the properties of the solution.
- **molar_fractions** (*numpy.array*) – The molar fractions of each endmember in the solution. Can be reset using the set_composition() method.

property name

Human-readable name of this material.

By default this will return the name of the class, but it can be set to an arbitrary string. Overriden in Mineral.

property endmembers

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state(*pressure*, *temperature*)

(copied from set_state):

Set the material to the given pressure and temperature.

Parameters

- **pressure** (*float*) – The desired pressure in [Pa].
- **temperature** (*float*) – The desired temperature in [K].

property formula

Returns molar chemical formula of the solution.

property activities

Returns a list of endmember activities [unitless].

property activity_coefficients

Returns a list of endmember activity coefficients (gamma = activity / ideal activity) [unitless].

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property partial_gibbs

Returns endmember partial molar Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial molar volumes [m^3/mol]. Property specific to solutions.

property partial_entropies

Returns endmember partial molar entropies [J/K/mol]. Property specific to solutions.

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs energy at constant pressure [J/mol]. Property specific to solutions.

property molar_helmholtz

Returns molar Helmholtz energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_mass

Returns molar mass of the solution [kg/mol].

property excess_pressure

Returns excess pressure of the solution [Pa]. Specific property for solutions.

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isothermal_compressibility_reuss

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property isentropic_compressibility_reuss

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property stoichiometric_matrix

A sympy Matrix where each element M[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property stoichiometric_array

An array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

property reaction_basis

An array where each element arr[i,j] corresponds to the number of moles of endmember[j] involved in reaction[i].

property n_reactions

The number of reactions in reaction_basis.

property independent_element_indices

A list of an independent set of element indices. If the amounts of these elements are known (element_amounts), the amounts of the other elements can be inferred by -compositional_null_basis[independent_element_indices].dot(element_amounts).

property dependent_element_indices

The element indices not included in the independent list.

property compositional_null_basis

An array N such that N.b = 0 for all bulk compositions that can be produced with a linear sum of the endmembers in the solution.

property endmember_formulae

A list of formulae for all the endmember in the solution.

property endmember_names

A list of names for all the endmember in the solution.

property n_endmembers

The number of endmembers in the solution.

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

property alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].

- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property gibbs

Extensive Gibbs free energy of the material (J).

property gr

Alias for *grueneisen_parameter()*

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

property internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property temperature

Returns current temperature that was set with *set_state()*.

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

`to_string()`

Returns the name of the mineral class

`unroll()`

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

`property v_p`

Alias for *p_wave_velocity()*

`property v_phi`

Alias for *bulk_sound_velocity()*

`property v_s`

Alias for *shear_wave_velocity()*

`property volume`

Extensive volume of the material (m^3).

`class burnman.AnisotropicSolution(name, solution_model, psi_excess_function, anisotropic_parameters, molar_fractions=None)`

Bases: *Solution*, *AnisotropicMineral*

A class implementing the anisotropic solution model described in [Myh24]. This class is derived from Solution and AnisotropicMineral, and inherits most of the methods from those classes.

Instantiation of an AnisotropicSolution is similar to that of a scalar Solution, except that each of the endmembers must be an instance of an AnisotropicMineral, and an additional function and parameter

dictionary are passed to the constructor of AnisotropicSolution that describe excess contributions to the anisotropic state tensor (Psi_xs) and its derivatives with respect to volume and temperature. The function arguments should be ln(V), Pth, p (a vector of proportions) and the parameter dictionary, in that order. The output variables Psi_xs_Voigt, dPsidf_Pth_Voigt_xs and dPsidPth_f_Voigt_xs (all 6x6 matrices) and dPsiIdp_xs (a 3 x 3 x n_endmember matrix) must be returned in that order in a tuple. States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.AnisotropicSolution`.

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].

compute_base_properties()

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters

molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

property ones

property eye

property depsdn_fixed_VT

Gradient in strain with respect to composition at fixed volume and temperature under hydrostatic conditions.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

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 K_T

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 K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

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 alpha

Alias for *thermal_expansivity()*

property beta_S

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider either using isentropic_compressibility_tensor, isentropic_compressibility_reuss or isentropic_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isentropic_compliance_tensor.

property beta_T

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider using isothermal_compressibility_tensor, isothermal_compressibility_reuss or isothermal_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isothermal_compliance_tensor.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property cell_parameters

Returns

The molar cell parameters of the mineral, given in standard form: [a, b, c, α, β, γ], where the first three floats are the lengths of the vectors in [m] defining the cell constructed from one mole of formula units. The last three floats are angles between vectors (given in radians). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (1D)

property cell_vectors

Returns

The vectors of the cell constructed from one mole of formula units [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (2D)

check_standard_parameters(*anisotropic_parameters*)

christoffel_tensor(*propagation_direction*)

Calculates the adiabatic Christoffel tensor.

The isentropic stiffness tensors provided in this and other anisotropic classes correspond to the isentropic derivative of stress with respect to symmetric infinitesimal strain. This corresponds to the isentropic elastic tensor B of Wallace (1967; doi:10.1103/PhysRev.162.776).

The Christoffel (acoustic) matrix is the contraction T_ik = S_ijkl n_j n_l, where S is Wallace's isentropic equation-of-motion tensor (not to be confused with the isentropic compliance tensor). For a unit direction n, contracting S gives T = sym(T_B) + ((n . sigma . n) I - sigma) / 2, where T_B_ik = B_ijkl n_j n_l and sigma is the tension-positive Cauchy stress. This derivation follows from Wallace (1967), using his Equations (2.24), (2.28), and (2.36).

Parameters

propagation_direction – Direction or array of directions with shape (..., 3), expressed in the same frame as the stress and stiffness tensors. Each direction is normalized before contraction.

Returns

Christoffel tensor in Pa, with shape (..., 3, 3).

Return type

numpy.ndarray

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type
numpy.ndarray

property compositional_null_basis

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector *b = [b_C, b_O]*, the constraint *1 * b_C - 2 * b_O = 0* must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property deformation_gradient_tensor

Returns
The deformation gradient tensor describing the deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature) to its current state. Note that this does not include the rotation of the mineral back into the crystal frame, which is described by the rotation_matrix property.

Return type
numpy.array (2D)

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property dependent_element_indices

The element indices not included in the independent list.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns

Effective shear modulus in [Pa].

Return type

float

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property endmember_names

A list of names for all the endmembers in the solution.

property endmembers

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property formula

Returns molar chemical formula of the solution. :rtype: Counter

property full_isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property full_isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property full_isothermal_compliance_tensor

Returns
The isothermal compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property full_isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property gibbs

Extensive Gibbs free energy of the material (J).

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property grueneisen_tensor

Returns

The grueneisen tensor [unitless]. This is defined by [BM67] as $\mathbb{C}_{\mathrm{N}ijkl}\alpha_{kl}V/C_P$.

Return type

numpy.array (2D)

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isentropic_bulk_modulus_voigt

Returns

The Voigt bound on the isentropic bulk modulus in [Pa].

Return type

float

property isentropic_bulk_modulus_vrh

Returns

The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property isentropic_compressibility_reuss

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property isentropic_compressibility_tensor

Returns
The isentropic compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property isentropic_compressibility_voigt

Returns
The Voigt bound on the isentropic compressibility in [1/Pa].

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the lateral response to loading.

Return type
float

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction relative to the stiffness tensor [1/Pa].

Return type
float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response directions relative to the stiffness tensor [unitless].

Return type
float

`isentropic_shear_modulus`(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor [Pa].

Return type
float

`property isentropic_shear_modulus_reuss`

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

`property isentropic_shear_modulus_voigt`

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

`property isentropic_shear_modulus_vrh`

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

`property isentropic_stiffness_tensor`

Returns
The isentropic stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

`property isentropic_thermal_gradient`

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

`property isentropic_universal_elastic_anisotropy`

Returns
The universal elastic anisotropy [unitless]

Return type
float

`isentropic_youngs_modulus`(*direction*)

Returns
The isentropic Youngs modulus in a given direction relative to the stiffness tensor [Pa].

Return type
float

`property isothermal_bulk_modulus_reuss`

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

`property isothermal_bulk_modulus_voigt`

Returns
The Voigt bound on the isothermal bulk modulus in [Pa].

Return type
float

`property isothermal_compliance_tensor`

Returns
The isothermal compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

`property isothermal_compressibility_reuss`

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

`property isothermal_compressibility_tensor`

Returns
The isothermal compressibility tensor [1/Pa].

Return type
numpy.array (2D)

`property isothermal_compressibility_voigt`

Returns
The Voigt bound on the isothermal compressibility in [1/Pa].

Return type
float

property isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{Tpq}$).

Return type
numpy.array (2D)

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [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_isometric_heat_capacity

Returns
The molar heat capacity at constant strain [J/K/mol].

Return type
float

property molar_mass

Returns molar mass of the solution [kg/mol].

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property n_endmembers

The number of endmembers in the solution.

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 number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns
An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type
numpy.ndarray

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

property rotation_matrix

Returns
The rotation from the unrotated frame into the crystallographic frame. Column vectors transform as `R @ v` and second-order tensors as `R @ A @ R.T`. For orthotropic minerals, this is the identity matrix.

Return type
numpy.array (2D)

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: 'bm3shear2', 'bm3', 'mgd2', 'mgd3', 'slb2', 'slb3', 'mt', 'hp_tmt', or 'cork'. Alternatively, you can pass a user defined class which derives from the equation_of_state base class. After calling set_method(), any existing derived properties (e.g., elastic parameters or thermodynamic potentials) will be out of date, so set_state() will need to be called again.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they

should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies
- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

standard_psi_function(*f*, *Pth*, *params*)

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

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 temperature

Returns current temperature that was set with *set_state()*.

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity
Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property thermal_expansivity_tensor

Returns
The tensor of thermal expansivities [1/K].

Return type
numpy.array (2D)

property thermal_stress_tensor

Returns
The change in stress with temperature at constant strain [Pa/K].

Return type
numpy.array (2D)

to_string()
Returns the name of the mineral class

unroll()
Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property unrotated_cell_vectors

Returns
The vectors of the cell [m] constructed from one mole of formula units after deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type
numpy.array (2D)

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

wave_velocities(*propagation_direction*)

Returns
The compressional wave velocity, and two shear wave velocities in a given propagation direction [m/s].

Return type
list, containing the wave speeds and directions of particle motion relative to the stiffness tensor

class burnman.**SolutionModel**

Bases: object

This is the base class for a solution model, intended for use in defining solutions and performing thermodynamic calculations on them. All minerals of type *burnman.Solution* use a solution model for defining how the endmembers in the solution interact.

A user wanting a new solution model should define the functions included in the base class. All of the functions in the base class return zero, so if the user-defined solution model does not implement them, they essentially have no effect, and the Gibbs free energy and molar volume of a solution will be equal to the weighted arithmetic averages of the different endmember values.

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess enthalpy of the solution.

Return type

float

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial Gibbs free energy of each endmember.

Return type

numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

`excess_partial_volumes`(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

`Cp_excess`(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

`alphaV_excess`(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

class burnman.**ElasticSolutionModel**

Bases: object

This is the base class for an Elastic solution model, intended for use in defining solutions and performing thermodynamic calculations on them. All minerals of type *burnman.Solution* use a solution model for defining how the endmembers in the solution interact.

A user wanting a new solution model should define the functions included in the base class. All of the functions in the base class return zero, so if the user-defined solution model does not implement them, they essentially have no effect, and the Helmholtz energy and pressure of a solution will be equal to the weighted arithmetic averages of the different endmember values.

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

20.10.2 Relaxed solutions

class burnman.**RelaxedSolution**(*solution*, *relaxation_vectors*, *unrelaxed_vectors*)

Bases: *Solution*

A class implementing a relaxed solution model. This class is derived from Solution, and inherits most of the methods from that class.

Instantiation of a RelaxedSolution involves passing an Solution, plus a set of vectors that represent rapid deformation modes. For example, a solution of MgO, FeHSO and FeLSO (high and low spin wuestite) can rapidly change proportion of high spin and low spin iron, and so a single vector should be passed: np.array([[0., -1., 1.]]) or some multiple thereof.

States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.RelaxedSolution`.

set_state(*pressure*, *temperature*, *relaxed=True*)

Sets the state of the solution. Also relaxes the structure parameters if set_composition() has already been used and if the relaxed argument has been set to True.

Parameters

- **pressure** (*float*) – The pressure of the solution [Pa]
- **temperature** (*float*) – The temperature of the solution [K]
- **relaxed** (*bool, optional*) – Whether to minimize the Gibbs energy of the material by changing the values of the structure parameters. Defaults to True.

set_composition(*molar_fractions*, *q_initial=None*, *relaxed=True*)

Sets the composition of the model. Also relaxes the structure parameters if set_state() has already been used and if the relaxed argument has been set to True.

Parameters

- **molar_fractions** (*1D numpy array*) – Molar fractions of the independent endmembers corresponding to the unrelaxed vectors specified during initialisation.
- **q_initial** (*1D numpy array, optional*) – Initial values of the structure parameters. Defaults to None, in which case the preexisting initial values are used (first set to 0.001 during initialisation).
- **relaxed** (*bool, optional*) – Whether to minimize the Gibbs energy of the material by changing the values of the structure parameters. Defaults to True.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding and Nelder-Mead minimization methods.

Composition should have been set before this function is used.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property dqdz_relaxed

The change of the structure parameters with respect to strain and temperature that minimizes the Gibbs energy.

property isothermal_compressibility_reuss

(copied from isothermal_compressibility_reuss):

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property thermal_expansivity

(copied from thermal_expansivity):

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.alpha.

property molar_heat_capacity_p

(copied from molar_heat_capacity_p):

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property molar_heat_capacity_v

(copied from molar_heat_capacity_v):

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property isentropic_bulk_modulus_reuss

(copied from isentropic_bulk_modulus_reuss):

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isentropic_compressibility_reuss

(copied from isentropic_compressibility_reuss):

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

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 alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type

numpy.ndarray

property compositional_null_basis

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector $b = [b_C, b_O]$, the constraint *1 * b_C - 2 * b_O = 0* must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property dependent_element_indices

The element indices not included in the independent list.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property endmember_names

A list of names for all the endmembers in the solution.

property endmembers

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property formula

Returns molar chemical formula of the solution. :rtype: Counter

property gibbs

Extensive Gibbs free energy of the material (J).

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [J/mol]. Aliased with self.helmholtz.

property molar_internal_energy

Returns molar internal energy of the mineral [J/mol]. Aliased with self.energy

property molar_mass

Returns molar mass of the solution [kg/mol].

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property n_endmembers

The number of endmembers in the solution.

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 number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns
An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type
numpy.ndarray

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 equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', '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_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies
- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

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 temperature

Returns current temperature that was set with *set_state()*.

Note

Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

to_string()

Returns the name of the mineral class

unroll()

Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns

A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type

tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

class burnman.**RelaxedAnisotropicSolution**(*anisotropic_solution*, *relaxation_vectors*, *unrelaxed_vectors*)

Bases: *AnisotropicSolution*

A class implementing the relaxed anisotropic solution model described in [Myh25b]. This class is derived from AnisotropicSolution, and inherits most of the methods from that class.

Instantiation of a RelaxedAnisotropicSolution involves passing an AnisotropicSolution, plus a set of vectors that represent rapid deformation modes. For example, a solution of MgO, FeHSO and FeLSO (high and low spin wuestite) can rapidly change proportion of high spin and low spin iron, and so a single vector should be passed: np.array([[0., -1., 1.]]) or some multiple thereof.

States of the mineral can only be queried after setting the pressure and temperature using set_state() and the composition using set_composition().

This class is available as `burnman.RelaxedAnisotropicSolution`.

set_state(*pressure*, *temperature*, *relaxed=True*)

Sets the state of the solution. Also relaxes the structure parameters if set_composition() has already been used and if the relaxed argument has been set to True.

Parameters

- **pressure** (*float*) – The pressure of the solution [Pa]
- **temperature** (*float*) – The temperature of the solution [K]
- **relaxed** (*bool, optional*) – Whether to minimize the Gibbs energy of the material by changing the values of the structure parameters. Defaults to True.

set_composition(*molar_fractions*, *q_initial=None*, *relaxed=True*)

Sets the composition of the model. Also relaxes the structure parameters if set_state() has already been used and if the relaxed argument has been set to True.

Parameters

- **molar_fractions** (*1D numpy array*) – Molar fractions of the independent endmembers corresponding to the unrelaxed vectors specified during initialisation.
- **q_initial** (*1D numpy array, optional*) – Initial values of the structure parameters. Defaults to None, in which case the preexisting initial values are used (first set to 0.001 during initialisation).
- **relaxed** (*bool, optional*) – Whether to minimize the Gibbs energy of the material by changing the values of the structure parameters. Defaults to True.

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure. In order to ensure self-consistency, this function does not use any pressure functions from the material classes, but instead finds the pressure using the brentq root-finding and Nelder-Mead minimization methods.

Composition should have been set before this function is used.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property dqdz_relaxed

The change of the structure parameters with respect to strain and temperature that minimizes the Helmholtz energy.

property isothermal_stiffness_tensor

(copied from isothermal_stiffness_tensor):

Returns
The isothermal stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

property thermal_stress_tensor

(copied from thermal_stress_tensor):

Returns
The change in stress with temperature at constant strain [Pa/K].

Return type
numpy.array (2D)

property molar_isometric_heat_capacity

(copied from molar_isometric_heat_capacity):

Returns
The molar heat capacity at constant strain [J/K/mol].

Return type
float

property isothermal_compliance_tensor

(copied from isothermal_compliance_tensor):

Returns
The isothermal compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{T}pq}$).

Return type
numpy.array (2D)

property full_isothermal_compliance_tensor

(copied from full_isothermal_compliance_tensor):

Returns
The isothermal compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property thermal_expansivity_tensor

(copied from thermal_expansivity_tensor):

Returns
The tensor of thermal expansivities [1/K].

Return type
numpy.array (2D)

property isothermal_compressibility_reuss

(copied from isothermal_compressibility_reuss):

Returns
The Reuss bound on the isothermal compressibility in [1/Pa].

Return type
float

property isothermal_bulk_modulus_reuss

(copied from isothermal_bulk_modulus_reuss): (copied from isothermal_bulk_modulus_reuss):

Returns isothermal bulk modulus of the material.

> **Note**
> Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
float

property isentropic_compressibility_reuss

(copied from isentropic_compressibility_reuss):

Returns
The Reuss bound on the isentropic compressibility in [1/Pa].

Return type
float

property isentropic_bulk_modulus_reuss

(copied from isentropic_bulk_modulus_reuss): (copied from isentropic_bulk_modulus_reuss):

Returns the adiabatic bulk modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
float

property thermal_expansivity

(copied from thermal_expansivity): (copied from thermal_expansivity):

Returns thermal expansion coefficient of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
float

property molar_heat_capacity_p

(copied from molar_heat_capacity_p): (copied from molar_heat_capacity_p):

Returns molar heat capacity at constant pressure of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
float

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

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 K_T

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 K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

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 alpha

Alias for *thermal_expansivity()*

property beta_S

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider either using isentropic_compressibility_tensor, isentropic_compressibility_reuss or isentropic_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isentropic_compliance_tensor.

property beta_T

Anisotropic minerals do not have a single isentropic compressibility. This function returns a NotImplementedError. Users should instead consider using isothermal_compressibility_tensor, isothermal_compressibility_reuss or isothermal_compressibility_voigt (all derived from AnisotropicMineral), or directly querying the elements in the isothermal_compliance_tensor.

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property cell_parameters

Returns

The molar cell parameters of the mineral, given in standard form: [a, b, c, α, β, γ], where the first three floats are the lengths of the vectors in [m] defining the cell constructed from one mole of formula units. The last three floats are angles between vectors (given in radians). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (1D)

property cell_vectors

Returns

The vectors of the cell constructed from one mole of formula units [m]. See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type

numpy.array (2D)

check_standard_parameters(*anisotropic_parameters*)

christoffel_tensor(*propagation_direction*)

Calculates the adiabatic Christoffel tensor.

The isentropic stiffness tensors provided in this and other anisotropic classes correspond to the isentropic derivative of stress with respect to symmetric infinitesimal strain. This corresponds to the isentropic elastic tensor B of Wallace (1967; doi:10.1103/PhysRev.162.776).

The Christoffel (acoustic) matrix is the contraction T_ik = S_ijkl n_j n_l, where S is Wallace's isentropic equation-of-motion tensor (not to be confused with the isentropic compliance tensor). For a unit direction n, contracting S gives T = sym(T_B) + ((n . sigma . n) I - sigma) / 2, where T_B_ik = B_ijkl n_j n_l and sigma is the tension-positive Cauchy stress. This derivation follows from Wallace (1967), using his Equations (2.24), (2.28), and (2.36).

Parameters

propagation_direction – Direction or array of directions with shape (..., 3), expressed in the same frame as the stress and stiffness tensors. Each direction is normalized before contraction.

Returns

Christoffel tensor in Pa, with shape (..., 3, 3).

Return type

numpy.ndarray

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type
numpy.ndarray

property compositional_null_basis

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector *b = [b_C, b_O]*, the constraint *1 * b_C - 2 * b_O = 0* must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

compute_base_properties()

copy()

debug_print(*indent=''*)

Print a human-readable representation of this Material.

property deformation_gradient_tensor

Returns
The deformation gradient tensor describing the deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature) to its current state. Note that this does not include the rotation of the mineral back into the crystal frame, which is described by the rotation_matrix property.

Return type
numpy.array (2D)

property density

Returns density of the solution [kg/m^3]. Aliased with self.rho.

property dependent_element_indices

The element indices not included in the independent list.

property depsdn_fixed_VT

Gradient in strain with respect to composition at fixed volume and temperature under hydrostatic conditions.

property effective_isentropic_bulk_modulus

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns

Effective isentropic bulk modulus in [Pa].

Return type

float

property effective_shear_modulus

Returns the effective shear modulus of the mineral.

> **Note**
>
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns

Effective shear modulus in [Pa].

Return type

float

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property endmember_names

A list of names for all the endmembers in the solution.

property endmembers

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type

list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns

List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or `numpy.array`, n-dimensional

property excess_enthalpy

Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property excess_entropy

Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property excess_gibbs

Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_entropies

Returns excess partial entropies [J/K]. Property specific to solutions.

property excess_partial_gibbs

Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes

Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_volume

Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property eye

property formula

Returns molar chemical formula of the solution. :rtype: Counter

property full_isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in standard form ($\mathbb{S}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property full_isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{N}ijkl}$).

Return type
numpy.array (4D)

property full_isothermal_stiffness_tensor

Returns
The isothermal stiffness tensor [Pa] in standard form ($\mathbb{C}_{\mathrm{T}ijkl}$).

Return type
numpy.array (4D)

property gibbs

Extensive Gibbs free energy of the material (J).

property gibbs_hessian

Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property gr

Alias for *grueneisen_parameter()*

property grueneisen_parameter

Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property grueneisen_tensor

Returns
The grueneisen tensor [unitless]. This is defined by [BM67] as $\mathbb{C}_{\mathrm{N}ijkl}\alpha_{kl}V/C_P$.

Return type
numpy.array (2D)

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_voigt

Returns
The Voigt bound on the isentropic bulk modulus in [Pa].

Return type
float

property isentropic_bulk_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic bulk modulus [Pa].

Return type
float

property isentropic_compliance_tensor

Returns
The isentropic compliance tensor [1/Pa] in Voigt form ($\mathbb{S}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property isentropic_compressibility_tensor

Returns
The isentropic compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property isentropic_compressibility_voigt

Returns
The Voigt bound on the isentropic compressibility in [1/Pa].

Return type
float

property isentropic_isotropic_poisson_ratio

Returns
The isotropic Poisson ratio (mu) [unitless]. A metric of the lateral response to loading.

Return type
float

isentropic_linear_compressibility(*direction*)

Returns
The linear isentropic compressibility in a given direction relative to the stiffness tensor [1/Pa].

Return type
float

isentropic_poissons_ratio(*axial_direction*, *lateral_direction*)

Returns
The isentropic poisson ratio given loading and response directions relative to the stiffness tensor [unitless].

Return type
float

isentropic_shear_modulus(*plane_normal*, *shear_direction*)

Returns
The isentropic shear modulus on a plane in a given shear direction relative to the stiffness tensor [Pa].

Return type
float

property isentropic_shear_modulus_reuss

Returns
The Reuss bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_voigt

Returns
The Voigt bound on the isentropic shear modulus [Pa].

Return type
float

property isentropic_shear_modulus_vrh

Returns
The Voigt-Reuss-Hill average of the isentropic shear modulus [Pa].

Return type
float

property isentropic_stiffness_tensor

Returns
The isentropic stiffness tensor [Pa] in Voigt form ($\mathbb{C}_{\mathrm{N}pq}$).

Return type
numpy.array (2D)

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property isentropic_universal_elastic_anisotropy

Returns
The universal elastic anisotropy [unitless]

Return type
float

isentropic_youngs_modulus(*direction*)

Returns
The isentropic Youngs modulus in a given direction relative to the stiffness tensor [Pa].

Return type
float

property isothermal_bulk_modulus_voigt

Returns
The Voigt bound on the isothermal bulk modulus in [Pa].

Return type
float

property isothermal_compressibility_tensor

Returns
The isothermal compressibility tensor [1/Pa].

Return type
numpy.array (2D)

property isothermal_compressibility_voigt

Returns
The Voigt bound on the isothermal compressibility in [1/Pa].

Return type
float

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [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 solution [kg/mol].

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property n_endmembers

The number of endmembers in the solution.

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 number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property ones

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns
An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type
numpy.ndarray

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

property rotation_matrix

Returns
The rotation from the unrotated frame into the crystallographic frame. Column vectors transform as `R @ v` and second-order tensors as `R @ A @ R.T`. For orthotropic minerals, this is the identity matrix.

Return type
numpy.array (2D)

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: 'bm3shear2', '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_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies
- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

standard_psi_function(*f*, *Pth*, *params*)

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

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 temperature

Returns current temperature that was set with *set_state()*.

> **Note**
>
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

`to_string()`
Returns the name of the mineral class

`unroll()`
Unroll this material into a list of *burnman.Mineral* and their molar fractions. All averaging schemes then operate on this list of minerals. Note that the return value of this function may depend on the current state (temperature, pressure).

> **Note**
>
> Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

`property unrotated_cell_vectors`

Returns
The vectors of the cell [m] constructed from one mole of formula units after deformation of the mineral from its undeformed state (i.e. the state at the reference pressure and temperature). See the documentation for the function *cell_parameters_to_vectors()* for the assumed relationships between the cell vectors and spatial coordinate axes.

Return type
numpy.array (2D)

`property v_p`
Alias for *p_wave_velocity()*

`property v_phi`
Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

wave_velocities(*propagation_direction*)

Returns

The compressional wave velocity, and two shear wave velocities in a given propagation direction [m/s].

Return type

list, containing the wave speeds and directions of particle motion relative to the stiffness tensor

20.10.3 Mechanical solution

class burnman.classes.solutionmodel.**MechanicalSolution**(*endmembers*)

Bases: *SolutionModel*

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of mixing; the total gibbs free energy of the solution is equal to the dot product of the molar gibbs free energies and molar fractions of the constituent materials.

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess Gibbs energy.

Return type

float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial Gibbs free energy of each endmember.

Return type

numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial volume of each endmember.

Return type

numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

alphaV_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticMechanicalSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

An extremely simple class representing a mechanical solution model. A mechanical solution experiences no interaction between endmembers. Therefore, unlike ideal solutions there is no entropy of mixing; the total Helmholtz energy of the solution is equal to the dot product of the molar Helmholtz energies and molar fractions of the constituent materials.

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of each endmember.

Return type

numpy.array

20.10.4 Ideal solution

class burnman.classes.solutionmodel.**IdealSolution**(*endmembers*)

Bases: *SolutionModel*

A solution model representing ideal mixing between endmembers. In ideal solutions, the enthalpy of mixing is equal to zero; i.e. chemical mixing of endmembers at fixed pressure and temperature does not involve the absorption or release of heat. The only contribution to the Gibbs energy of mixing is the configurational entropy S_{conf}:

$$\mathcal{G}_{\text{mixing}} = -T\Delta S_{\text{conf}},$$
$$S_{\text{conf}} = Rx_c^s \ln \frac{x_c^s}{\sum_{c^s} x_c^s}$$

where s is a site in the lattice, c are the species mixing on site s. x_c^s is the absolute number of species c on site s in the lattice; it is calculated by multiplying the proportion of the species on the site by the multiplicity of the site per formula unit and the number of moles of formula units. This class allows the site multiplicities to vary linearly between endmembers. This is known as a Temkin model [Tem45].

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

configurational_entropy(*molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

property ones

A vector of ones with length equal to the number of endmembers :return: ones :rtype: 1D numpy array

property eye

An identity matrix with size equal to the number of endmembers :return: eye :rtype: 2D numpy array

property eyeones

A convenience function consisting of two concatenations of an identity matrix and ones vector with size equal to the number of endmembers. :return: delta_ij 1_k + delta_ik 1_j :rtype: 3D numpy array

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

alphaV_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

class burnman.classes.elasticsolutionmodel.**ElasticIdealSolution**(*endmembers*)

Bases: *ElasticSolutionModel*

A class representing an ideal solution model. Calculates the excess Helmholtz energy and entropy due to configurational entropy. Excess internal energy and volume are equal to zero.

The multiplicity of each type of site in the structure is allowed to change linearly as a function of endmember proportions. This class is therefore equivalent to the entropic part of a Temkin-type model [Tem45].

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]

- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

configurational_entropy(*molar_fractions*)

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of the solution.

Return type

float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess Helmholtz energy.

Return type

float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess pressure of the solution.

Return type

float

20.10.5 Asymmetric regular solution

class burnman.classes.solutionmodel.**AsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *IdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Gibbs energy is given by the expression [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.

Parameters

- **endmembers** (*list of tuples*) – A list of tuples defining the endmembers of the solution. Each tuple contains a *burnman.Mineral* object and a string defining the site formula of that endmember.
- **alphas** (*list of floats*) – A list of van Laar parameters governing asymmetry in the excess properties.
- **energy_interaction** (*list of lists of floats*) – A 2D list of interaction parameters defining the energetic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **volume_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the volumetric interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **entropy_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the entropic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

alphaV_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

`configurational_entropy`(*molar_fractions*)

`excess_enthalpy`(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

`excess_entropy`(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

`excess_gibbs_energy`(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].

- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

property eye

An identity matrix with size equal to the number of endmembers :return: eye :rtype: 2D numpy array

property eyeones

A convenience function consisting of two concatenations of an identity matrix and ones vector with size equal to the number of endmembers. :return: delta_ij 1_k + delta_ik 1_j :rtype: 3D numpy array

property ones

A vector of ones with length equal to the number of endmembers :return: ones :rtype: 1D numpy array

class burnman.classes.elasticsolutionmodel.**ElasticAsymmetricRegularSolution**(*endmembers*, *alphas*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*

Bases: *ElasticIdealSolution*

Solution model implementing the asymmetric regular solution model formulation as described in [HollandPowell03].

The excess nonconfigurational Helmholtz energy is given by the expression:

$$\mathcal{F}_{\text{excess}} = \alpha^T p(\phi^T W \phi)$$

α is a vector of van Laar parameters governing asymmetry in the excess properties.

$$\phi_i = \frac{\alpha_i p_i}{\sum_{k=1}^{n} \alpha_k p_k}, W_{ij} = \frac{2w_{ij}}{\alpha_i + \alpha_j} \text{for i<j}$$

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess Helmholtz energy of each endmember

Return type

numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of each endmember.

Return type

numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess pressure of each endmember.

Return type

numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

configurational_entropy(*molar_fractions*)

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of the solution.

Return type

float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]

- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

20.10.6 Symmetric regular solution

class burnman.classes.solutionmodel.**SymmetricRegularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*)

Bases: *AsymmetricRegularSolution*

Solution model implementing the Symmetric Regular Solution Model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class where all van Laar parameters are equal to unity.

The excess terms in the Symmetric Regular Solution Model 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.

Parameters

- **endmembers** (*list of tuples*) – A list of tuples defining the endmembers of the solution. Each tuple contains a *burnman.Mineral* object and a string defining the site formula of that endmember.

- **energy_interaction** (*list of lists of floats*) – A 2D list of interaction parameters defining the energetic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **volume_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the volumetric interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.
- **entropy_interaction** (*list of lists of floats*) – (optional) A 2D list of interaction parameters defining the entropic interactions between endmembers. The list should be upper-triangular, such that the first row has n-1 elements (for n endmembers), the second row has n-2 elements, etc.

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

activities(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

alphaV_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

configurational_entropy(*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 model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].

- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

property eye

An identity matrix with size equal to the number of endmembers :return: eye :rtype: 2D numpy array

property eyeones

A convenience function consisting of two concatenations of an identity matrix and ones vector with size equal to the number of endmembers. :return: delta_ij 1_k + delta_ik 1_j :rtype: 3D numpy array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

property ones

A vector of ones with length equal to the number of endmembers :return: ones :rtype: 1D numpy array

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

class burnman.classes.elasticsolutionmodel.**ElasticSymmetricRegularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*)

Bases: *ElasticAsymmetricRegularSolution*

Solution model implementing the symmetric regular solution model. This is a special case of the burnman.solutionmodel.AsymmetricRegularSolution class.

configurational_entropy(*molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess entropy of the solution.

Return type

float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

excess_pressure(*volume*, *temperature*, *molar_fractions*)
Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

20.10.7 Subregular solution

class burnman.classes.solutionmodel.**SubregularSolution**(*endmembers*, *energy_interaction*, *volume_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *volume_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *IdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess nonconfigurational Gibbs energy is given by the expression:

$$\mathcal{G}_{\text{excess}} = \sum_i \sum_{j>i} (p_i p_j^2 W_{ij} + p_j p_i^2 W_{ji} + \sum_{k>j>i} p_i p_j p_k W_{ijk})$$

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **volume_interaction** (*list of list of lists*) – The binary endmember interaction volumes. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **volume_ternary_terms** (*list of lists*) – The ternary interaction volumes. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns

The excess partial Gibbs free energy of each endmember.

Return type

numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

`VoverKT_excess`(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

`alphaV_excess`(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

`configurational_entropy`(*molar_fractions*)

`excess_enthalpy`(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].

- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

property eye

An identity matrix with size equal to the number of endmembers :return: eye :rtype: 2D numpy array

property eyeones

A convenience function consisting of two concatenations of an identity matrix and ones vector with size equal to the number of endmembers. :return: delta_ij 1_k + delta_ik 1_j :rtype: 3D numpy array

property ones

A vector of ones with length equal to the number of endmembers :return: ones :rtype: 1D numpy array

class burnman.classes.elasticsolutionmodel.**ElasticSubregularSolution**(*endmembers*, *energy_interaction*, *pressure_interaction=None*, *entropy_interaction=None*, *energy_ternary_terms=None*, *pressure_ternary_terms=None*, *entropy_ternary_terms=None*)

Bases: *ElasticIdealSolution*

Solution model implementing the subregular solution model formulation as described in [HW89]. The excess conconfigurational Helmholtz energy is given by the expression:

$$\mathcal{F}_{\text{excess}} = \sum_i \sum_{j>i} (p_i p_j^2 W_{ij} + p_j p_i^2 W_{ji} + \sum_{k>j>i} p_i p_j p_k W_{ijk})$$

Interaction parameters are inserted into a 3D interaction matrix during initialization to make use of numpy vector algebra.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.

- **energy_interaction** (*list of list of lists*) – The binary endmember interaction energies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **pressure_interaction** (*list of list of lists*) – The binary endmember interaction pressures. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **entropy_interaction** (*list of list of lists*) – The binary endmember interaction entropies. Each interaction[i, j-i-1, 0] corresponds to W(i,j), while interaction[i, j-i-1, 1] corresponds to W(j,i).
- **energy_ternary_terms** (*list of lists*) – The ternary interaction energies. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **pressure_ternary_terms** (*list of lists*) – The ternary interaction pressures. Each list should contain four entries: the indices i, j, k and the value of the interaction.
- **entropy_ternary_terms** (*list of lists*) – The ternary interaction entropies. Each list should contain four entries: the indices i, j, k and the value of the interaction.

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns

The excess Helmholtz energy of each endmember

Return type

numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]

- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

configurational_entropy(*molar_fractions*)

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

20.10.8 Function solution

class burnman.classes.solutionmodel.**FunctionSolution**(*endmembers*, *excess_gibbs_function*)

Bases: *IdealSolution*

Solution model implementing a generalized solution model. The extensive excess nonconfigurational Gibbs energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Gibbs energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.

- **excess_gibbs_function** (*function*) – The nonconfigurational Gibbs energy function with arguments pressure, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Gibbs energy must also double.

excess_partial_volumes(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume for each endmember of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial volume of each endmember.

Return type
numpy.array

Cp_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess isobaric heat capacity of the solution.

Return type
float

alphaV_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess alpha*V of the solution model

Return type
float

VoverKT_excess(*pressure*, *temperature*, *molar_fractions*)

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess V/K_T of the solution model

Return type
float

volume_hessian(*pressure*, *temperature*, *molar_fractions*)

excess_partial_gibbs_energies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy for each endmember of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial Gibbs free energy of each endmember.

Return type
numpy.array

excess_partial_entropies(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].

- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess partial entropy of each endmember.

Return type
numpy.array

gibbs_hessian(*pressure*, *temperature*, *molar_fractions*)

entropy_hessian(*pressure*, *temperature*, *molar_fractions*)

activity_coefficients(*pressure*, *temperature*, *molar_fractions*)

activities(*pressure*, *temperature*, *molar_fractions*)

configurational_entropy(*molar_fractions*)

excess_enthalpy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess enthalpy of the solution. The base class implementation assumes that the excess enthalpy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess enthalpy of the solution.

Return type
float

excess_entropy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess entropy of the solution.

Return type
float

excess_gibbs_energy(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Gibbs free energy of the solution. The base class implementation assumes that the excess gibbs free energy is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess Gibbs energy.

Return type
float

excess_volume(*pressure*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess volume of the solution. The base class implementation assumes that the excess volume is zero.

Parameters

- **pressure** (*float*) – Pressure at which to evaluate the solution model [Pa].
- **temperature** (*float*) – Temperature at which to evaluate the solution model [K].
- **molar_fractions** (*list of floats*) – List of molar fractions of the different independent endmembers in the solution model.

Returns
The excess volume of the solution.

Return type
float

property eye

An identity matrix with size equal to the number of endmembers :return: eye :rtype: 2D numpy array

property eyeones

A convenience function consisting of two concatenations of an identity matrix and ones vector with size equal to the number of endmembers. :return: delta_ij 1_k + delta_ik 1_j :rtype: 3D numpy array

property ones

A vector of ones with length equal to the number of endmembers :return: ones :rtype: 1D numpy array

class burnman.classes.elasticsolutionmodel.**ElasticFunctionSolution**(*endmembers*, *excess_helmholtz_function*)

Bases: *ElasticIdealSolution*

Solution model implementing a generalized elastic solution model. The extensive excess nonconfigurational Helmholtz energy is provided as a function by the user.

Derivatives are calculated using the autograd module, and so the user-defined excess Helmholtz energy function should be defined using autograd-friendly expressions.

Parameters

- **endmembers** (*list of lists*) – A list of all the independent endmembers in the solution. The first item of each list gives the Mineral object corresponding to the endmember. The second item gives the site-species formula.
- **excess_helmholtz_function** (*function*) – The nonconfigurational Helmholtz energy function with arguments volume, temperature and molar_amounts, in that order. Note that the function must be extensive; if the molar amounts are doubled, the Helmholtz energy must also double.

excess_partial_pressures(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure for each endmember of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of each endmember.

Return type
numpy.array

pressure_hessian(*volume*, *temperature*, *molar_fractions*)

excess_partial_helmholtz_energies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz energy for each endmember of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy of each endmember

Return type
numpy.array

excess_partial_entropies(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy for each endmember of the solution. The base class implementation assumes that the excess entropy is zero (true for mechanical solutions).

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of each endmember.

Return type
numpy.array

configurational_entropy(*molar_fractions*)

excess_entropy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess entropy of the solution. The base class implementation assumes that the excess entropy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess entropy of the solution.

Return type
float

excess_helmholtz_energy(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess Helmholtz free energy of the solution. The base class implementation assumes that the excess Helmholtz energy is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]

- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess Helmholtz energy.

Return type
float

excess_pressure(*volume*, *temperature*, *molar_fractions*)

Given a list of molar fractions of different phases, compute the excess pressure of the solution. The base class implementation assumes that the excess pressure is zero.

Parameters

- **volume** (*float*) – Volume at which to evaluate the solution model. [m^3/mol]
- **temperature** (*float*) – Temperature at which to evaluate the solution. [K]
- **molar_fractions** (*list of floats*) – List of molar fractions of the different endmembers in solution.

Returns
The excess pressure of the solution.

Return type
float

helmholtz_hessian(*volume*, *temperature*, *molar_fractions*)

entropy_hessian(*volume*, *temperature*, *molar_fractions*)

20.11 Solution tools

burnman.tools.solution.**transform_solution_to_new_basis**(*solution*, *new_basis*, *n_mbrs=None*, *solution_name=None*, *endmember_names=None*, *molar_fractions=None*)

Transforms a solution model from one endmember basis to another. Returns a new Solution object.

Parameters

- **solution** (*burnman.Solution* object) – The original solution object.
- **new_basis** (*2D numpy array*) – The new endmember basis, given as amounts of the old endmembers.
- **n_mbrs** (*float, optional*) – The number of endmembers in the new solution (defaults to the length of new_basis).
- **solution_name** (*str, optional*) – A name corresponding to the new solution.
- **endmember_names** (*list of str, optional*) – A list corresponding to the names of the new endmembers.

- **molar_fractions** (*numpy.array, optional*) – Fractions of the new end-members in the new solution.

Returns
The transformed solution.

Return type
burnman.Solution object

20.12 Compositions

20.12.1 Base class

class burnman.**Composition**(*composition_dictionary*, *unit_type='mass'*, *normalize=False*)

Bases: object

The Composition class can be used to store, modify and renormalize compositions, and also convert between mass, molar and atomic amounts. Weight is provided as an alias for mass, as we assume that only Earthlings will use this software.

Composition can parse dictionaries of components, rather than only elements. Components are given as strings (e.g. 'MgSiO3', 'Fe2O3', etc). The Composition class breaks these components down into their elemental constituents internally.

Example:

```
from burnman.composition import Composition

comp_dict = {'MgSiO3': 0.5, 'FeSiO3': 0.5}
comp = Composition(comp_dict, unit_type='molar', normalize=True)
print(comp.atomic_composition)

# Renormalize to 1 kg of material
comp.renormalize(unit_type='mass', normalization_component='total',␣
↪normalization_amount=1.0)
print(comp.mass_composition)

# Add 0.1 moles of Al2O3 to the composition
comp.add_components({'Al2O3': 0.1}, unit_type='molar')
print(comp.molar_composition)

# Change the component set to only oxides
comp.change_component_set(['MgO', 'SiO2', 'FeO', 'Al2O3'])
print(comp.molar_composition)

# Remove components with less than 1.e-10 moles
comp.remove_null_components(tol=1.e-10)
print(comp.molar_composition)
```

(continues on next page)

(continued from previous page)

```
# Pretty-print the molar composition, normalized to 100 moles total
comp.print(unit_type='molar', significant_figures=3, normalization_
↪component='total', normalization_amount=100.0)
```

Parameters

- **composition_dictionary** (*dictionary*) – Dictionary of components (given as a string) and their amounts.
- **unit_type** (*str*) – 'mass', 'weight' or 'molar' (optional, 'mass' as default) Specify whether the input composition is given as mass or molar amounts.
- **normalize** (*bool*) – If False, absolute numbers of kilograms/moles of component are stored, otherwise the component amounts of returned compositions will sum to one (until Composition.renormalize() is used).

renormalize(*unit_type*, *normalization_component*, *normalization_amount*)

Change the total amount of material in the composition to satisfy a given normalization condition (mass, weight, molar, or atomic)

Parameters

- **unit_type** (*str*) – 'mass', 'weight', 'molar' or 'atomic' Unit type with which to normalize the composition
- **normalization_component** (*str*) – Component/element on which to renormalize. String must either be one of the components/elements already in the composition, or have the value 'total'.
- **normalization_amount** (*float*) – Amount of component in the renormalised composition.

add_components(*composition_dictionary*, *unit_type*)

Add (or remove) components from the composition. The components are added to the current state of the (mass, weight or molar) composition; if the composition has been renormalised, then this should be taken into account.

Parameters

- **composition_dictionary** (*dictionary*) – Components to add, and their amounts.
- **unit_type** (*str*) – 'mass', 'weight' or 'molar'. Unit type of the components to be added.

change_component_set(*new_component_list*)

Change the set of basis components without changing the bulk composition.

Will raise an exception if the new component set is invalid for the given composition.

Parameters

new_component_list (*list of strings*) – New set of basis components.

remove_null_components(*tol=1e-12*)

Removes components with absolute concentrations less than a given tolerance.

Parameters
tol (*float, optional*) – zero tolerance, defaults to 1.e-12

property weight_composition

An alias for mass composition [kg].

property molar_composition

Returns the molar composition as a counter [moles]

property atomic_composition

Returns the atomic composition as a counter [moles]

composition(*unit_type*)

Helper function to return the composition in the desired type.

Parameters
unit_type (*str*) – One of 'mass', 'weight', 'molar' and 'atomic'.

Returns
Mass (weight), molar or atomic composition.

Return type
OrderedCounter

print(*unit_type*, *significant_figures=1*, *normalization_component='total'*, *normalization_amount=None*)

Pretty-print function for the composition This does not renormalize the Composition object itself, only the printed values.

Parameters

- **unit_type** (*str*) – 'mass', 'weight', 'molar' or 'atomic' Unit type in which to print the composition.
- **significant_figures** (*int*) – Number of significant figures for each amount.
- **normalization_component** (*str*) – Component/element on which to renormalize. String must either be one of the components/elements already in composite, or have the value 'total'. (default = 'total')
- **normalization_amount** (*float*) – Amount of component in the renormalised composition. If not explicitly set, no renormalization will be applied.

20.12.2 Utility functions

burnman.classes.composition.**file_to_composition_list**(*fname*, *unit_type*, *normalize*)

Takes an input file with a specific format and returns a list of compositions (and associated comments) contained in that file.

Parameters

- **fname** (*str*) – Path to ascii file containing composition data. Lines beginning with a hash are not read. The first read-line of the datafile contains a list of tab or space-separated components (e.g. FeO or SiO2), followed by the word Comment.

  Following lines are lists of floats with the amounts of each component. After the component amounts, the user can write anything they like in the Comment section.

- **unit_type** (*str*) – 'mass', 'weight' or 'molar' Specify whether the compositions in the file are given as mass (weight) or molar amounts.

- **normalize** (*bool*) – 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).

Returns
A list of Composition objects and a list of comments.

Return type
list of *Composition*, list of list of str

20.12.3 Fitting functions

burnman.optimize.composition_fitting.**fit_composition_to_solution**(*solution*, *fitted_variables*, *variable_values*, *variable_covariances*, *variable_conversions=None*, *normalize=True*)

Fit a set of molar fractions to a solution model that best matches observed compositional variables (in a least-squares sense).

The fitting applies appropriate non-negativity constraints (i.e. no species can have a negative occupancy on a site).

Parameters

- **solution** (*burnman.Solution*) – The solution to use in the fitting procedure.

- **fitted_variables** (*list of str*) – A list of the variables used to find the best-fit molar fractions of the solution. These should either be elements such as "Fe", site_species such as "Fef_B" which would correspond to a species labelled Fef on the second site, or user-defined variables which are arithmetic sums of elements and/or site_species defined in "variable_conversions".

- **variable_values** (*numpy.array*) – Numerical values of the fitted variables. These should be given as amounts; they do not need to be normalized.

- **variable_covariances** (*2D numpy.array*) – Covariance matrix of the variables.

- **variable_conversions** (*dict of dict, or None*) – A dictionary converting any user-defined variables into an arithmetic sum of element and site-species amounts. For example, {'Mg_equal': {'Mg_A': 1., 'Mg_B': -1.}}, coupled with

Mg_equal = 0 would impose a constraint that the amount of Mg would be equal on the first and second site in the solution.

- **normalize** (*bool, default=True*) – Whether to normalize the molar fractions to sum to one.

Returns

(molar_fractions, covariance_matrix, weighted_residual):

- molar_fractions: Optimized molar fractions of each endmember.
- **covariance_matrix: Covariance matrix of the optimized molar** fractions, computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing** the square root of the weighted sum of squared differences between observed and modeled variables.

Return type

tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

burnman.optimize.composition_fitting.**fit_phase_proportions_to_bulk_composition**(*phase_compositions*, *bulk_composition*)

Determine the proportions of phases that best fit a given bulk composition (in a weighted constrained least squares sense).

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

(phase_proportions, covariance_matrix, weighted_residual)

- phase_proportions: Optimized proportions of each phase.
- **covariance_matrix: Covariance matrix of the optimized phase proportions,** computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing the square root** of the weighted sum of squared differences between observed and modeled bulk

compositions.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

20.13 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.

20.13.1 Base class

class burnman.**MaterialPolytope**(*equalities*, *inequalities*, *return_fractions=False*, *independent_endmember_occupancies=None*)

Bases: object

A class for creating and manipulating polytope objects using pycddlib.

This class is available as burnman.polytope.MaterialPolytope.

set_return_type(*return_fractions=False*)

Sets the return_type for the polytope object. Also deletes the cached endmember_occupancies property.

Parameters
return_fractions (*bool*) – Choose whether the generated polytope object should return fractions or floats.

property raw_vertices

Returns a list of the vertices of the polytope without any postprocessing. See also endmember_occupancies.

property limits

Return the limits of the polytope (the set of bounding inequalities).

property n_endmembers

Return the number of endmembers (the number of vertices of the polytope).

property endmember_occupancies

Return the endmember occupancies (a processed list of all of the vertex locations).

property independent_endmember_occupancies

Return an independent set of endmember occupancies (a linearly-independent set of vertex locations)

property endmembers_as_independent_endmember_amounts

Return a list of all the endmembers as a linear sum of the independent endmembers.

property independent_endmember_polytope

Returns the polytope expressed in terms of proportions of the independent endmembers. The polytope involves the first n-1 independent endmembers. The last endmember proportion makes the sum equal to one.

property independent_endmember_limits

Gets the limits of the polytope as a function of the independent endmembers.

subpolytope_from_independent_endmember_limits(*limits*)

Returns a smaller polytope by applying additional limits to the amounts of the independent endmembers.

subpolytope_from_site_occupancy_limits(*limits*)

Returns a smaller polytope by applying additional limits to the individual site occupancies.

grid(*points_per_edge=2*, *unique_sorted=True*, *grid_type='independent endmember proportions'*, *limits=None*)

Create a grid of points which span the polytope.

Parameters

- **points_per_edge** (*int*) – Number of points per edge of the polytope.
- **unique_sorted** (*bool*) – The gridding is done by splitting the polytope into a set of simplices. This means that points will be duplicated along vertices, faces etc. If unique_sorted is True, this function will sort and make the points unique. This is an expensive operation for large polytopes, and may not always be necessary.
- **grid_type** (*str*) – Whether to grid the polytope in terms of independent endmember proportions or site occupancies. Choices are 'independent endmember proportions' or 'site occupancies'
- **limits** (*numpy.array (2D)*) – Additional inequalities restricting the gridded area of the polytope.

Returns

A list of points gridding the polytope.

Return type

numpy.array (2D)

20.13.2 Polytope tools

burnman.tools.polytope.**solution_polytope_from_charge_balance**(*charges*, *charge_total*, *return_fractions=False*)

Creates a polytope object from a list of the charges for each species on each site and the total charge for all site-species.

Parameters

- **charges** (*2D list of floats*) – 2D list containing the total charge for species j on site i, including the site multiplicity. So, for example, a solution with the site formula [Mg,Fe]3[Mg,Al,Si]2Si3O12 would have the following list: [[6., 6.], [4., 6., 8.]].
- **charge_total** (*float*) – The total charge for all site-species per formula unit. The example given above would have charge_total = 12.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns

A polytope object corresponding to the parameters provided.

Return type

burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**solution_polytope_from_endmember_occupancies**(*endmember_occupancies*, *return_fractions=False*)

Creates a polytope object from a list of independent endmember occupancies.

Parameters

- **endmember_occupancies** (*2D numpy array*) – 2D list containing the site-species occupancies j for endmember i. So, for example, a solution with independent endmembers [Mg]3[Al]2Si3O12, [Mg]3[Mg0.5Si0.5]2Si3O12, [Fe]3[Al]2Si3O12 might have the following array: [[1., 0., 1., 0., 0.], [1., 0., 0., 0.5, 0.5], [0., 1., 1., 0., 0.]], where the order of site-species is Mg_A, Fe_A, Al_B, Mg_B, Si_B.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns

A polytope object corresponding to the parameters provided.

Return type

burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**composite_polytope_at_constrained_composition**(*composite*, *composition*, *return_fractions=False*)

Creates a polytope object from a Composite object and a composition. This polytope describes the complete set of valid composite endmember amounts that satisfy the compositional constraints.

Parameters

- **composite** (*burnman.Composite* object) – A composite containing one or more Solution and Mineral objects.
- **composition** (*dict*) – A dictionary containing the amounts of each element.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns

A polytope object corresponding to the parameters provided.

Return type

burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**reorder_to_echelon**(*A*)

Reorders the rows of a matrix A so that it is in echelon form. :param A: The input matrix. :type A: 2D numpy array :returns: The reordered matrix. :rtype: 2D numpy array

burnman.tools.polytope.**simplify_composite_with_composition**(*composite*, *composition*)

Takes a composite and a composition, and returns the simplest composite object that spans the solution space at the given composition.

For example, if the composition is given as {'Mg': 2., 'Si': 1.5, 'O': 5.}, and the composite is given as a mix of Mg,Fe olivine and pyroxene solutions, this function will return a composite that only contains the Mg-bearing endmembers.

If the solutions have endmember proportions that are consistent with the bulk composition, the site occupancies of the new solution models are set to the values in the original models.

Parameters

- **composite** (*burnman.Composite* object) – The initial Composite object.
- **composition** (*dict*) – A dictionary containing the amounts of each element.

Returns

The simplified Composite object

Return type

burnman.Composite object

burnman.tools.polytope.**greedy_independent_endmember_selection**(*endmember_site_occupancies*, *site_occupancies*, *small_fraction_tol=0.0*, *norm_tol=1e-12*)

Greedy algorithm to select independent endmembers from a solution to approximate given site occupancies through a non-negative linear combination of endmember site occupancies.

This function starts with the full site occupancies and then iteratively selects endmembers to approximate those site occupancies. It loops through all possible endmembers in a number of steps. At each step the algorithm selects the endmember that can be subtracted in the largest amount from the current residual site occupancies without making any site occupancy negative. The process continues until either no endmember can be subtracted in an amount greater than fraction_tol, or the norm of the residual site occupancies is less than tol.

Parameters

- **endmember_site_occupancies** (*np.ndarray*) – A 2D array of shape (m, n), where m is the number of endmembers and n is the number of sites. Each row corresponds to the site occupancies of an endmember.
- **site_occupancies** (*np.ndarray*) – A 1D array of length n, representing the target site occupancies to approximate.
- **small_fraction_tol** (*float, optional, default 0.0*) – Algorithm stops if no endmember can be added with a molar fraction larger than this value.
- **norm_tol** (*float, optional, default 1e-12*) – Algorithm stops if the norm of the residual site occupancies is less than this value.

Returns

indices of selected endmembers, their fractions, and the final residual site occupancies.

Return type

tuple(list[int], list[float], np.ndarray)

20.14 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.

20.14.1 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

The average bulk modulus K. $[Pa]$

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli G for a composite. This defines the interface for this method, and is not implemented in the base class.

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli of each phase in the composite $[Pa]$.

Returns

The average shear modulus G. $[Pa]$

Return type

float

20.14.2 Voigt bound

class burnman.averaging_schemes.**Voigt**

Bases: *AveragingScheme*

Class for computing the Voigt (iso-strain) bound for elastic properties. This derives from `burnman.averaging_schemes.averaging_scheme`, and implements the `burnman.averaging_schemes.averaging_scheme.average_bulk_moduli()` and `burnman.averaging_schemes.averaging_scheme.average_shear_moduli()` functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite K with the Voigt (iso-strain) bound, given by:

$$K_V = \Sigma_i V_i K_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns
The Voigt average bulk modulus K_R $[Pa]$.

Return type
float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the Voigt (iso-strain) bound, given by:

$$G_V = \Sigma_i V_i G_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns
The Voigt average shear modulus G_V $[Pa]$.

Return type
float

20.14.3 Reuss bound

class burnman.averaging_schemes.**Reuss**

Bases: *AveragingScheme*

Class for computing the Reuss (iso-stress) bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the Reuss (iso-stress) bound, given by:

$$K_R = \left(\Sigma_i \frac{V_i}{K_i} \right)^{-1}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns
The Reuss average bulk modulus K_R $[Pa]$.

Return type
float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the Reuss (iso-stress) bound, given by:

$$G_R = \left(\Sigma_i \frac{V_i}{G_i} \right)^{-1}$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns
The Reuss average shear modulus G_R $[Pa]$.

Return type
float

20.14.4 Voigt-Reuss-Hill average

class burnman.averaging_schemes.**VoigtReussHill**

Bases: *AveragingScheme*

Class for computing the Voigt-Reuss-Hill average for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the Voigt-Reuss-Hill average, given by:

$$K_{VRH} = \frac{K_V + K_R}{2}$$

This is simply a shorthand for an arithmetic average of the bounds given by burnman.averaging_schemes.voigt and burnman.averaging_schemes.reuss.

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$. Not used in this average.

Returns

The Voigt-Reuss-Hill average bulk modulus K_{VRH} $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli G of a composite with the Voigt-Reuss-Hill average, given by:

$$G_{VRH} = \frac{G_V + G_R}{2}$$

This is simply a shorthand for an arithmetic average of the bounds given by burnman.averaging_schemes.voigt and burnman.averaging_schemes.reuss.

$$G_V = \Sigma_i V_i G_i$$

Parameters

- **volumes** (*list of floats*) – List of the volume of each phase in the composite $[m^3]$.
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite $[Pa]$. Not used in this average.
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite $[Pa]$.

Returns

The Voigt-Reuss-Hill average shear modulus G_{VRH} $[Pa]$.

Return type

float

20.14.5 Hashin-Shtrikman upper bound

class burnman.averaging_schemes.**HashinShtrikmanUpper**

Bases: *AveragingScheme*

Class for computing the upper Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss

bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the upper Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The upper Hashin-Shtrikman average bulk modulus K. $[Pa]$

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the upper Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The upper Hashin-Shtrikman average shear modulus G. $[Pa]$

Return type

float

20.14.6 Hashin-Shtrikman lower bound

class burnman.averaging_schemes.**HashinShtrikmanLower**

Bases: *AveragingScheme*

Class for computing the lower Hashin-Shtrikman bound for elastic properties. This derives from burnman.averaging_schemes.averaging_scheme, and implements the burnman.averaging_schemes.averaging_scheme.average_bulk_moduli() and burnman.averaging_schemes.averaging_scheme.average_shear_moduli() functions. Implements Formulas from [WDOConnell76]. The Hashin-Shtrikman bounds are tighter than the Voigt and Reuss

bounds because they make the additional assumption that the orientation of the phases are statistically isotropic. In some cases this may be a good assumption, and in others it may not be.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the lower Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The lower Hashin-Shtrikman average bulk modulus K. $[Pa]$

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the shear moduli of a composite with the lower Hashin-Shtrikman bound. Implements Formulas from [WDOConnell76], which are too lengthy to reproduce here.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The lower Hashin-Shtrikman average shear modulus G. $[Pa]$

Return type

float

20.14.7 Hashin-Shtrikman arithmetic average

class burnman.averaging_schemes.**HashinShtrikmanAverage**

Bases: *AveragingScheme*

Class for computing arithmetic mean of the Hashin-Shtrikman bounds on elastic properties. This derives from `burnman.averaging_schemes.averaging_scheme`, and implements the `burnman.averaging_schemes.averaging_scheme.average_bulk_moduli()` and `burnman.averaging_schemes.averaging_scheme.average_shear_moduli()` functions.

average_bulk_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the arithmetic mean of the upper and lower Hashin-Shtrikman bounds.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. $[m^3]$
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. Not used in this average. $[Pa]$

Returns

The arithmetic mean of the Hashin-Shtrikman bounds on bulk modulus K $[Pa]$.

Return type

float

average_shear_moduli(*volumes*, *bulk_moduli*, *shear_moduli*)

Average the bulk moduli of a composite with the arithmetic mean of the upper and lower Hashin-Shtrikman bounds.

Parameters

- **volumes** (*list of floats*) – List of the volumes of each phase in the composite. [m^3].
- **bulk_moduli** (*list of floats*) – List of bulk moduli K of each phase in the composite. Not used in this average. $[Pa]$
- **shear_moduli** (*list of floats*) – List of shear moduli G of each phase in the composite. $[Pa]$

Returns

The arithmetic mean of the Hashin-Shtrikman bounds on shear modulus G $[Pa]$.

Return type

float

20.15 Geotherms

class burnman.classes.geotherm.**Geotherm**(*depths_array*, *temperatures_array*)

Bases: object

Base class for all geotherms. Subclasses should implement a temperatures(depths) method or inherit the one provided here.

Parameters

- **depths_array** (*np.array*) – Array of depths (in meters).
- **temperatures_array** (*np.array*) – Array of temperatures (in Kelvin).

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**BrownShankland**

Bases: *Geotherm*

Geotherm from Brown and Shankland, 1981 ([BS81])

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**Anderson**

Bases: *Geotherm*

Geotherm from Anderson, 1982 ([And82])

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**StaceyContinental**

Bases: *Geotherm*

Continental geotherm from Stacey, 1977 ([Sta77])

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**StaceyOceanic**

Bases: *Geotherm*

Oceanic geotherm from Stacey, 1977 ([Sta77])

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**Katsura2022**

Bases: *Geotherm*

Geotherm from Katsura, 2022 ([Kat22])

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**Plesa2022Mars6cm3**

Bases: *Geotherm*

Martian areotherm from Plesa, 2022 ([PWK+22]) assuming 6 cm^3/mol activation volume for mantle viscosity.

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.array of float

Raises

- **ValueError** – If depths are outside the range of the geotherm.
- **ValueError** – If depths_array or temperatures_array is None.

class burnman.classes.geotherm.**GeothermFromPressures**(*pressures_array*, *temperatures_array*, *depth_to_pressure_model=<burnman.classes.seismic.object>*)

Bases: *Geotherm*

Geotherm defined by pressures and temperatures.

Parameters

- **pressures_array** (*np.array*) – Array of pressures (in Pa).
- **temperatures_array** (*np.array*) – Array of temperatures (in Kelvin).
- **depth_to_pressure_model** (burnman.seismic.SeismicModel or other object with a pressure(depth) method.) – Object with an object.pressure(depth) method, such as burnman.seismic.SeismicModel. If not given, the PREM model will be used.

temperatures_from_pressures(*pressures*)

Evaluate the geotherm at given pressures.

Parameters
pressures (*array-like of float*) – Pressures at which to evaluate the geotherm (in Pa).

Returns
Temperatures at the given pressures (in Kelvin).

Return type
np.ndarray of float

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.ndarray of float

class burnman.classes.geotherm.**Anzellini2013**(*depth_to_pressure_model=<burnman.classes.seismic.PREM object>*)

Bases: *GeothermFromPressures*

Geotherm from Anzellini et al., 2013 ([ADM+13]) (their Figure S4). Mantle part is from Steinberger and Holme, 2008 ([SH08]).

Parameters
depth_to_pressure_model (burnman.seismic.SeismicModel or other object with a pressure(depth) method.) – Object with an object.pressure(depth) method, such as burnman.seismic.SeismicModel. If not given, the PREM model will be used.

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.ndarray of float

`temperatures_from_pressures`(*pressures*)

Evaluate the geotherm at given pressures.

Parameters
pressures (*array-like of float*) – Pressures at which to evaluate the geotherm (in Pa).

Returns
Temperatures at the given pressures (in Kelvin).

Return type
np.ndarray of float

class burnman.classes.geotherm.**AdiabaticGeotherm**(*material*, *T_0*, *P_0*, *depth_to_pressure_model=<burnman.classes.seismic.PREM object>*)

Bases: *GeothermFromPressures*

Geotherm calculated as an adiabat for a given material, anchored on a specified temperature and pressure.

In addition to the temperature method of the standard Geotherm class, this class also provides a temperatures_from_pressures method.

Initialisation parameters:

Parameters

- **material** (burnman.Material) – *burnman.Material* object.
- **T_0** (*float*) – Temperature at the anchor pressure [K].
- **P_0** (*float*) – The anchor pressure [Pa].
- **depth_to_pressure_model** (burnman.seismic.SeismicModel or other object with a pressure(depth) method.) – Object with an object.pressure(depth) method, such as burnman.seismic.SeismicModel. If not given, the PREM model will be used.

`temperatures_from_pressures`(*pressures*)

Evaluate the adiabat geotherm at given pressures.

Parameters
pressures (*array-like of float*) – Pressures at which to evaluate the geotherm (in Pa).

Returns
Temperatures at the given pressures (in Kelvin).

Return type
np.ndarray of float

temperatures(*depths*)

Evaluate the geotherm at given depths.

Parameters
depths (*array-like of float*) – Depths at which to evaluate the geotherm (in meters).

Returns
Temperatures at the given depths (in Kelvin).

Return type
np.ndarray of float

burnman.classes.geotherm.**adiabatic_profile**(*pressures*, *rock*, *T_0*, *P_0=None*)

This function calculates an adiabat for a rock anchored on a specified temperature and pressure.

A starting guess is provided by integrating:

$$\frac{\partial T}{\partial P} = \frac{\gamma T}{K_s}$$

from the anchor point. In this expression, γ is the Grueneisen parameter and K_s is the adiabatic bulk modulus. The exact adiabatic profile is then found by finding the temperature profile along which the entropy is constant.

Parameters

- **pressures** (*list of floats*) – The list of pressures in $[Pa]$ at which to evaluate the geotherm.
- **rock** (burnman.composite) – Composite for which we compute the adiabat. From this material we must compute average Grueneisen parameters and adiabatic bulk moduli for each pressure/temperature.
- **T_0** (*float*) – An anchor temperature, corresponding to the temperature at the anchor pressure. $[K]$
- **P_0** (*float, optional*) – An anchor pressure. If not given, the first pressure in the pressures list will be used.

Returns
The list of temperatures for each pressure. $[K]$

Return type
numpy.array of floats

20.16 Layers and Planets

20.16.1 Layer

class burnman.Layer(*name=None*, *radii=None*, *verbose=False*)

Bases: object

The base class for a planetary layer. The user needs to set the following before properties can be computed:

- set_material(), which sets the material of the layer, e.g. a mineral, solid_solution, or composite
- set_temperature_mode(), either predefine, or set to an adiabatic profile
- set_pressure_mode(), to set the self-consistent pressure (with user-defined option the pressures can be overwritten). To set the self-consistent pressure the pressure at the top and the gravity at the bottom of the layer need to be set.
- make(), computes the self-consistent part of the layer and starts the settings to compute properties within the layer

Note that the entire planet this layer sits in is not necessarily self-consistent, as the pressure at the top of the layer is a function of the density within the layer (through the gravity). Entire planets can be computed self-consistently with the planet class. Properties will be returned at the pre-defined radius array, although the evaluate() function can take a newly defined depthlist and values are interpolated between these (sufficient sampling of the layer is needed for this to be accurate).

reset()

Resets all cached material properties. It is typically not required for the user to call this function.

set_material(*material*)

Set the material of a Layer with a Material

set_temperature_mode(*temperature_mode='adiabatic'*, *temperatures=None*, *temperature_top=None*)

Sets temperatures within the layer as user-defined values or as a (potentially perturbed) adiabat.

Parameters

- **temperature_mode** (*string*) – This can be set to 'user-defined', 'adiabatic', or 'perturbed-adiabatic'. 'user-defined' fixes the temperature with the profile input by the user. 'adiabatic' self-consistently computes the adiabat when setting the state of the layer. 'perturbed-adiabatic' adds the user input array to the adiabat. This allows the user to apply boundary layers (for example).
- **temperatures** (*array of float*) – The desired fixed temperatures in [K]. Should have same length as defined radii in layer.
- **temperature_top** (*float*) – Temperature at the top of the layer. Used if the temperature mode is chosen to be 'adiabatic' or 'perturbed-adiabatic'. If 'perturbed-adiabatic' is chosen as the temperature mode, temperature_top corresponds to the true temperature at the top of the layer, and the reference isentrope at this radius is defined to lie at a temperature of temperature_top - temperatures[-1].

set_pressure_mode(*pressure_mode='self-consistent'*, *pressures=None*, *gravity_bottom=None*, *pressure_top=None*, *n_max_iterations=50*, *max_delta=1e-05*)

Sets the pressure mode of the layer, which can either be 'user-defined', or 'self-consistent'.

Parameters

- **pressure_mode** (*string*) – This can be set to 'user-defined' or 'self-consistent'. 'user-defined' fixes the pressures with the profile input by the user in the 'pressures' argument. 'self-consistent' forces Layer to calculate pressures self-consistently. If this is selected, the user will need to supply values for the gravity_bottom [m/s^2] and pressure_top [Pa] arguments.
- **pressures** (*array of floats*) – Pressures [Pa] to set layer to (if the 'user-defined' pressure_mode has been selected). The array should be the same length as the layers user-defined radii array.
- **pressure_top** (*float*) – Pressure [Pa] at the top of the layer.
- **gravity_bottom** (*float*) – Gravity [m/s^2] at the bottom of the layer.
- **n_max_iterations** (*integer*) – Maximum number of iterations to reach self-consistent pressures.
- **max_delta** (*float*) – Relative update to the highest pressure in the layer between iterations to stop iterations.

make()

This routine needs to be called before evaluating any properties. If pressures and temperatures are not user-defined, they are computed here. This method also initializes an array of copied materials from which properties can be computed.

evaluate(*properties*, *radlist=None*, *radius_planet=None*)

Function that is used to evaluate properties across the layer. If radlist is not defined, values are returned at the internal radlist. If asking for different radii than the internal radlist, pressure and temperature values are interpolated and the layer material evaluated at those pressures and temperatures.

Parameters

- **properties** (*list of strings*) – List of properties to evaluate.
- **radlist** (*array of floats*) – Radii to evaluate properties at. If left empty, internal radii list is used.
- **planet_radius** (*float*) – Planet outer radius. Used only to calculate depth.

Returns

1D or 2D array of requested properties (1D if only one property was requested)

Return type

numpy.array

property mass

Calculates the mass of the layer [kg]

property moment_of_inertia

Returns the moment of inertia of the layer [kg m^2]

property gravity

Returns gravity profile of the layer [m s^(-2)]

property bullen

Returns the Bullen parameter across the layer. The Bullen parameter assess if compression as a function of pressure is like homogeneous, adiabatic compression. Bullen parameter =1 , homogeneous, adiabatic compression Bullen parameter > 1 , more compressed with pressure, e.g. across phase transitions Bullen parameter < 1, less compressed with pressure, e.g. across a boundary layer.

property brunt_vasala

Returns the brunt-vasala (or buoyancy) frequency, N, across the layer. This frequency assess the stabilty of the layer: N < 0, fluid will convect N= 0, fluid is neutral N > 0, fluid is stabily stratified.

property pressure

Returns current pressures across the layer that was set with *set_state()*.

Aliased with *P()*.

Returns
Pressures in [Pa] at the predefined radii.

Return type
numpy.array

property temperature

Returns current temperature across the layer that was set with *set_state()*.

- Aliased with *T()*.

Returns
Temperatures in [K] at the predefined radii.

Return type
numpy.array

property molar_internal_energy

Returns the molar internal energies across the layer.

Notes

- Needs to be implemented in derived classes.
- Aliased with `energy()`.

Returns
The internal energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_gibbs

Returns the molar Gibbs free energies across the layer.

Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_helmholtz

Returns the molar Helmholtz free energies across the layer.

Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz energies in [J/mol] at the predefined radii.

Return type
numpy.array

property molar_mass

Returns molar mass of the layer.

Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
numpy.array

property molar_volume

Returns molar volumes across the layer.

Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volumes in [m^3/mol] at the predefined radii.

Return type
numpy.array

property density

Returns the densities across this layer.

Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The densities of this material in [kg/m^3] at the predefined radii.

Return type
numpy.array

property molar_entropy

Returns molar entropies acroos the layer.

Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropies in [J/K/mol] at the predefined radii.

Return type
numpy.array

property molar_enthalpy

Returns molar enthalpies across the layer.

Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpies in [J/mol] at the predefined radii.

Return type
numpy.array

property isothermal_bulk_modulus_reuss

Returns isothermal bulk moduli across the layer.

Notes

- Needs to be implemented in derived classes.
- Aliased with *K_T()*.

Returns
Bulk moduli in [Pa] at the predefined radii.

Return type
numpy.array

property isentropic_bulk_modulus_reuss

Returns the adiabatic bulk moduli across the layer.

Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa] at the predefined radii.

Return type
numpy.array

property isothermal_compressibility_reuss

Returns isothermal compressibilities across the layer (or inverse isothermal bulk moduli).

Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibilities in [1/Pa] at the predefined radii.

Return type
numpy.array

property isentropic_compressibility_reuss

Returns adiabatic compressibilities across the layer (or inverse adiabatic bulk moduli).

Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibilities in [1/Pa] at the predefined radii.

Return type
numpy.array

property shear_modulus

Returns shear moduli across the layer.

Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear moduli in [Pa] at the predefined radii.

Return type
numpy.array

property p_wave_velocity

Returns P wave speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speeds in [m/s] at the predefined radii.

Return type
numpy.array

property bulk_sound_velocity

Returns bulk sound speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocities in [m/s] at the predefined radii.

Return type
numpy.array

property shear_wave_velocity

Returns shear wave speeds across the layer.

Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speeds in [m/s] at the predefined radii.

Return type
numpy.array

property grueneisen_parameter

Returns the grueneisen parameters across the layer.

Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameters [unitless] at the predefined radii.

Return type
numpy.array

property thermal_expansivity

Returns thermal expansion coefficients across the layer.

Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivities in [1/K] at the predefined radii.

Return type
numpy.array

property molar_heat_capacity_v

Returns molar heat capacity at constant volumes across the layer.

Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Heat capacities in [J/K/mol] at the predefined radii.

Return type
numpy.array

property molar_heat_capacity_p

Returns molar_heat capacity at constant pressures across the layer.

Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Heat capacities in [J/K/mol] at the predefined radii.

Return type
numpy.array

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property energy

Alias for *molar_internal_energy()*

property helmholtz

Alias for *molar_helmholtz()*

property gibbs

Alias for *molar_gibbs()*

property V

Alias for *molar_volume()*

property rho

Alias for *density()*

property S

Alias for *molar_entropy()*

property H

Alias for *molar_enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property G

Alias for *shear_modulus()*

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property gr

Alias for *grueneisen_parameter()*

property alpha

Alias for *thermal_expansivity()*

property C_v

Alias for *molar_heat_capacity_v()*

property C_p

Alias for *molar_heat_capacity_p()*

class burnman.**BoundaryLayerPerturbation**(*radius_bottom*, *radius_top*, *rayleigh_number*, *temperature_change*, *boundary_layer_ratio*)

Bases: object

A class that implements a temperature perturbation model corresponding to a simple thermal boundary layer. The model takes the following form: T = a*exp((r - r1)/(r0 - r1)*c) + b*exp((r - r0)/(r1 - r0)*c) The relationships between the input parameters and a, b and c are given below.

This model is a simpler version of that proposed by [RM81].

Parameters

- **radius_bottom** (*float*) – The radius at the bottom of the layer (r0) [m].
- **radius_top** (*float*) – The radius at the top of the layer (r1) [m].
- **rayleigh_number** (*float*) – The Rayleigh number of convection within the layer. The exponential scale factor is this number to the power of 1/4 (Ra = c^4).
- **temperature_change** (*float*) – The total difference in potential temperature across the layer [K]. temperature_change = (a + b)*exp(c).
- **boundary_layer_ratio** (*float*) – The ratio of the linear scale factors (a/b) corresponding to the thermal boundary layers at the top and bottom of the layer. A number greater than 1 implies a larger change in temperature across the top boundary than the bottom boundary.

temperature(*radii*)

Returns the temperature at one or more radii [K].

Parameters
radii (*float or numpy.array*) – The radii at which to evaluate the temperature.

Returns
The temperatures at the requested radii.

Return type
float or numpy.array

dTdr(*radii*)

Returns the thermal gradient at one or more radii [K/m].

Parameters
radii (*float or numpy.array*) – The radii at which to evaluate the thermal gradients.

Returns
The thermal gradient at the requested radii.

Return type
float or numpy.array

set_model_thermal_gradients(*dTdr_bottom*, *dTdr_top*)

Reparameterizes the model based on the thermal gradients at the bottom and top of the model.

Parameters

- **dTdr_bottom** (*float*) – The thermal gradient at the bottom of the model [K/m]. Typically negative for a cooling planet.
- **dTdr_top** (*float*) – The thermal gradient at the top of the model [K/m]. Typically negative for a cooling planet.

20.16.2 Planet

class burnman.**Planet**(*name*, *layers*, *n_max_iterations=50*, *max_delta=1e-05*, *verbose=False*)

Bases: object

A class to build (self-consistent) Planets made out of Layers (*burnman.Layer*). By default the planet is set to be self-consistent (with zero pressure at the surface and zero gravity at the center), but this can be overwritte using the set_pressure_mode(). Pressure_modes defined in the individual layers will be ignored. If temperature modes are already set for each of the layers, when the planet is initialized, the planet will be built immediately.

reset()

Resets all cached material properties. It is typically not required for the user to call this function.

get_layer(*name*)

Returns a layer with a given name

Parameters
name (*str*) – Given name of a layer

Returns
Layer with the given name.

Return type
burnman.Layer

get_layer_by_radius(*radius*)

Returns a layer in which this radius lies

Parameters
radius (*float*) – Radius at which to evaluate the layer.

Returns
Layer in which the radius lies.

Return type
burnman.Layer

evaluate(*properties*, *radlist=None*)

Function that is generally used to evaluate properties of the different layers and stitch them together. If asking for different radii than the internal radlist, pressure and temperature values are interpolated and the layer material evaluated at those pressures and temperatures.

Parameters

- **properties** (*list of strings*) – List of properties to evaluate

- **radlist** (*array of floats*) – Radii to evaluate properties at. If left empty, internal radius lists are used.

Returns
1D or 2D array of requested properties (1D if only one property was requested)

Return type
numpy.array

set_pressure_mode(*pressure_mode='self-consistent'*, *pressures=None*, *pressure_top=0.0*, *gravity_bottom=0.0*, *n_max_iterations=50*, *max_delta=1e-05*)

Sets the pressure mode of the planet by user-defined values are in a self-consistent fashion. pressure_mode is 'user-defined' or 'self-consistent'. The default for the planet is self-consistent, with zero pressure at the surface and zero pressure at the center.

Parameters

- **pressure_mode** (*str*) – This can be set to 'user-defined' or 'self-consistent'.
- **pressures** (*array of floats*) – Pressures (Pa) to set layer to ('user-defined'). This should be the same length as defined radius array for the layer.
- **pressure_top** (*float*) – Pressure (Pa) at the top of the layer.
- **gravity_bottom** (*float*) – Gravity (m/s^2) at the bottom the layer
- **n_max_iterations** (*int*) – Maximum number of iterations to reach self-consistent pressures.

make()

This routine needs to be called before evaluating any properties. If pressures and temperatures are self-consistent, they are computed across the planet here. Also initializes an array of materials in each Layer to compute properties from.

property mass

calculates the mass of the entire planet [kg]

property average_density

calculates the average density of the entire planet [kg/m^3]

property moment_of_inertia

#Returns the moment of inertia of the planet [kg m^2]

property moment_of_inertia_factor

#Returns the moment of inertia of the planet [kg m^2]

property depth

Returns depth of the layer [m]

property gravity

Returns gravity of the layer [m s^(-2)]

property bullen

Returns the Bullen parameter

property brunt_vasala

property pressure

Returns current pressure that was set with *set_state()*.

Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
array of floats

property temperature

Returns current temperature that was set with *set_state()*.

Aliased with *T()*.

Returns
Temperature in [K].

Return type
array of floats

property molar_internal_energy

Returns the molar internal energy of the planet.

Needs to be implemented in derived classes. Aliased with `energy()`.

Returns
The internal energy in [J/mol].

Return type
array of floats

property molar_gibbs

Returns the molar Gibbs free energy of the planet.

Needs to be implemented in derived classes. Aliased with *gibbs()*.

Returns
Gibbs energy in [J/mol].

Return type
array of floats

property molar_helmholtz

Returns the molar Helmholtz free energy of the planet.

Needs to be implemented in derived classes. Aliased with *helmholtz()*.

Returns
Helmholtz energy in [J/mol].

Return type
array of floats

property molar_mass

Returns molar mass of the planet.

Needs to be implemented in derived classes.

Returns
Molar mass in [kg/mol].

Return type
array of floats

property molar_volume

Returns molar volume of the planet.

Needs to be implemented in derived classes. Aliased with *V()*.

Returns
Molar volume in [m^3/mol].

Return type
array of floats

property density

Returns the density of this planet.

Needs to be implemented in derived classes. Aliased with *rho()*.

Returns
The density of this material in [kg/m^3].

Return type
array of floats

property molar_entropy

Returns molar entropy of the planet.

Needs to be implemented in derived classes. Aliased with *S()*.

Returns
Entropy in [J/K/mol].

Return type
array of floats

property molar_enthalpy

Returns molar enthalpy of the planet.

Needs to be implemented in derived classes. Aliased with *H()*.

Returns
Enthalpy in [J/mol].

Return type
array of floats

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the planet.

Needs to be implemented in derived classes. Aliased with *K_T()*.

Returns
Isothermal bulk modulus in [Pa].

Return type
array of floats

property isentropic_bulk_modulus_reuss

Returns the adiabatic bulk modulus of the planet.

Needs to be implemented in derived classes. Aliased with *K_S()*.

Returns
Adiabatic bulk modulus in [Pa].

Return type
array of floats

property isothermal_compressibility_reuss

Returns isothermal compressibility of the planet (or inverse isothermal bulk modulus).

Needs to be implemented in derived classes. Aliased with *beta_T()*.

Returns
Isothermal compressibility in [1/Pa].

Return type
array of floats

property isentropic_compressibility_reuss

Returns adiabatic compressibility of the planet (or inverse adiabatic bulk modulus).

Needs to be implemented in derived classes. Aliased with *beta_S()*.

Returns
Adiabatic compressibility in [1/Pa].

Return type
array of floats

property shear_modulus

Returns shear modulus of the planet.

Needs to be implemented in derived classes. Aliased with `beta_G()`.

Returns
Shear modulus in [Pa].

Return type
array of floats

property p_wave_velocity

Returns P wave speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_p()*.

Returns
P wave speed in [m/s].

Return type
array of floats

property bulk_sound_velocity

Returns bulk sound speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_phi()*.

Returns
Bulk sound velocity in [m/s].

Return type
array of floats

property shear_wave_velocity

Returns shear wave speed of the planet.

Needs to be implemented in derived classes. Aliased with *v_s()*.

Returns
Shear wave speed in [m/s].

Return type
array of floats

property grueneisen_parameter

Returns the grueneisen parameter of the planet.

Needs to be implemented in derived classes. Aliased with *gr()*.

Returns
Grueneisen parameters [unitless].

Return type
array of floats

property thermal_expansivity

Returns thermal expansion coefficient of the planet.

Needs to be implemented in derived classes. Aliased with *alpha()*.

Returns
Thermal expansivity in [1/K].

Return type
array of floats

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the planet.

Needs to be implemented in derived classes. Aliased with *C_v()*.

Returns
Isochoric heat capacity in [J/K/mol].

Return type
array of floats

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the planet.

Needs to be implemented in derived classes. Aliased with *C_p()*.

Returns
Isobaric heat capacity in [J/K/mol].

Return type
array of floats

property P

Alias for *pressure()*

property T

Alias for *temperature()*

property energy

Alias for *molar_internal_energy()*

property helmholtz

Alias for *molar_helmholtz()*

property gibbs

Alias for *molar_gibbs()*

property V

Alias for *molar_volume()*

property rho

Alias for *density()*

property S

Alias for *molar_entropy()*

property H

Alias for *molar_enthalpy()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

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()*

20.17 Thermodynamics

Burnman has a number of functions and classes which deal with the thermodynamics of single phases and aggregates.

20.17.1 Lattice Vibrations

20.17.1.1 Debye model

burnman.eos.debye.**debye_fn**(*x*)

Evaluate the Debye function. Takes the parameter xi = Debye_T/T

burnman.eos.debye.**debye_fn_cheb**(*x*)

Evaluate the Debye function using a Chebyshev series expansion coupled with asymptotic solutions of the function. Shamelessly adapted from the GSL implementation of the same function (Itself adapted from Collected Algorithms from ACM). Should give the same result as debye_fn(x) to near machine-precision.

burnman.eos.debye.**thermal_energy**(*T*, *debye_T*, *n*)

calculate the thermal energy of a substance. Takes the temperature, the Debye temperature, and n, the number of atoms per molecule. Returns thermal energy in J/mol

burnman.eos.debye.**molar_heat_capacity_v**(*T*, *debye_T*, *n*)

Heat capacity at constant volume. In J/K/mol

burnman.eos.debye.**helmholtz_energy**(*T*, *debye_T*, *n*)

Helmholtz free energy of lattice vibrations in the Debye model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyways.

burnman.eos.debye.**entropy**(*T*, *debye_T*, *n*)

Entropy due to lattice vibrations in the Debye model [J/K].

burnman.eos.debye.**dmolar_heat_capacity_v_dT**(*T*, *debye_T*, *n*)

First temperature derivative of the heat capacity at constant volume [J/K^2/mol].

burnman.eos.debye.**dhelmholtz_dTheta**(*T*, *debye_T*, *n*)

First derivative of the Helmholtz free energy with respect to the Debye temperature [J/K].

burnman.eos.debye.**d2helmholtz_dTheta2**(*T*, *debye_T*, *n*)

Second derivative of the Helmholtz free energy with respect to the Debye temperature [J/K^2].

burnman.eos.debye.**dentropy_dTheta**(*T*, *debye_T*, *n*)

First derivative of the entropy with respect to the Debye temperature [J/K].

20.17.1.2 Einstein model

burnman.eos.einstein.**thermal_energy**(*T*, *einstein_T*, *n*)

calculate the thermal energy of a substance. Takes the temperature, the Einstein temperature, and n, the number of atoms per molecule. Returns thermal energy in J/mol

burnman.eos.einstein.**molar_heat_capacity_v**(*T*, *einstein_T*, *n*)

Heat capacity at constant volume. In J/K/mol

burnman.eos.einstein.**helmholtz_energy**(*T*, *einstein_T*, *n*)

Helmholtz free energy of lattice vibrations in the Einstein model [J]. It is important to note that this does NOT include the zero point energy for the lattice. As long as you are calculating relative differences in F, this should cancel anyway.

burnman.eos.einstein.**entropy**(*T*, *einstein_T*, *n*)

Entropy due to lattice vibrations in the Einstein model [J/K]

burnman.eos.einstein.**dmolar_heat_capacity_v_dT**(*T*, *einstein_T*, *n*)

First temperature derivative of the heat capacity at constant volume according to the Einstein model [J/K^2/mol].

burnman.eos.einstein.**dentropy_dTheta**(*T*, *einstein_T*, *n*)

Derivative of Einstein entropy with respect to Einstein temperature [J/K^2]

burnman.eos.einstein.**dhelmholtz_dTheta**(*T*, *einstein_T*, *n*)

First derivative of Helmholtz free energy with respect to Einstein temperature [J/K]

burnman.eos.einstein.**d2helmholtz_dTheta2**(*T*, *einstein_T*, *n*)

Second derivative of Helmholtz free energy with respect to Einstein temperature [J/K^2]

20.17.2 Chemistry parsing and thermodynamics

burnman.utils.chemistry.**read_masses**()

A simple function to read a file with a two column list of elements and their masses into a dictionary

burnman.utils.chemistry.**atomic_masses = {'Ag': 0.107868, 'Al': 0.0269815, 'Ar': 0.039948, 'As': 0.0749216, 'Au': 0.196967, 'B': 0.010811, 'Ba': 0.137327, 'Be': 0.00901218, 'Bi': 0.20898, 'Br': 0.079904, 'C': 0.0120107, 'Ca': 0.040078, 'Cd': 0.112411, 'Ce': 0.140116, 'Cl': 0.035453, 'Co': 0.0589332, 'Cr': 0.0519961, 'Cs': 0.132905, 'Cu': 0.063546, 'Dy': 0.1625, 'Er': 0.167259, 'Eu': 0.151964, 'F': 0.0189984, 'Fe': 0.055845, 'Ga': 0.069723, 'Gd': 0.15725, 'Ge': 0.07264, 'H': 0.00100794, 'He': 0.0040026, 'Hf': 0.17849, 'Hg': 0.20059, 'Ho': 0.16493, 'I': 0.126904, 'In': 0.114818, 'Ir': 0.192217, 'K': 0.0390983, 'Kr': 0.083798, 'La': 0.138905, 'Li': 0.006941, 'Lu': 0.174967, 'Mg': 0.024305, 'Mn': 0.054938, 'Mo': 0.09596, 'N': 0.0140067, 'Na': 0.0229898, 'Nb': 0.0929064, 'Nd': 0.144242, 'Ne': 0.0201797, 'Ni': 0.0586934, 'O': 0.0159994, 'Os': 0.19023, 'P': 0.0309738, 'Pa': 0.231036, 'Pb': 0.2072, 'Pd': 0.10642, 'Pr': 0.140908, 'Pt': 0.195084, 'Rb': 0.0854678, 'Re': 0.186207, 'Rh': 0.102905, 'Ru': 0.10107, 'S': 0.032065, 'Sb': 0.12176, 'Sc': 0.0449559, 'Se': 0.07896, 'Si': 0.0280855, 'Sm': 0.15036, 'Sn': 0.11871, 'Sr': 0.08762, 'Ta': 0.180948, 'Tb': 0.158925, 'Te': 0.1276, 'Th': 0.232038, 'Ti': 0.047867, 'Tl': 0.204383, 'Tm': 0.168934, 'U': 0.238029, 'V': 0.0509415, 'Vc': 0.0, 'W': 0.18384, 'Xe': 0.131293, 'Y': 0.0889058, 'Yb': 0.173054, 'Zn': 0.06538, 'Zr': 0.091224}**

atomic_masses is a dictionary of atomic masses

burnman.utils.chemistry.**IUPAC_element_order = ['v', 'Og', 'Rn', 'Xe', 'Kr', 'Ar', 'Ne', 'He', 'Fr', 'Cs', 'Rb', 'K', 'Na', 'Li', 'Ra', 'Ba', 'Sr', 'Ca', 'Mg', 'Be', 'Lr', 'No', 'Md', 'Fm', 'Es', 'Cf', 'Bk', 'Cm', 'Am', 'Pu', 'Np', 'U', 'Pa', 'Th', 'Ac', 'Lu', 'Yb', 'Tm', 'Er', 'Ho', 'Dy', 'Tb', 'Gd', 'Eu', 'Sm', 'Pm', 'Nd', 'Pr', 'Ce', 'La', 'Y', 'Sc', 'Rf', 'Hf', 'Zr', 'Ti', 'Db', 'Ta', 'Nb', 'V', 'Sg', 'W', 'Mo', 'Cr', 'Bh', 'Re', 'Tc', 'Mn', 'Hs', 'Os', 'Ru', 'Fe', 'Mt', 'Ir', 'Rh', 'Co', 'Ds', 'Pt', 'Pd', 'Ni', 'Rg', 'Au', 'Ag', 'Cu', 'Cn', 'Hg', 'Cd', 'Zn', 'Nh', 'Tl', 'In', 'Ga', 'Al', 'B', 'Fl', 'Pb', 'Sn', 'Ge', 'Si', 'C', 'Mc', 'Bi', 'Sb', 'As', 'P', 'N', 'Lv', 'Po', 'Te', 'Se', 'S', 'O', 'H', 'Ts', 'At', 'I', 'Br', 'Cl', 'F']**

IUPAC_element_order is a list of all the elements. Element order is based loosely on electronegativity, following the scheme suggested by IUPAC, except that H comes after the Group 16 elements, not before them.

burnman.utils.chemistry.**dictionarize_formula**(*formula*)

A function to read a chemical formula string and convert it into a dictionary

Parameters

formula (*str*) – Chemical formula, written in the XnYm format, where the formula has n atoms of element X and m atoms of element Y

Returns

The same chemical formula, but expressed as a dictionary.

Return type

dict

burnman.utils.chemistry.**sum_formulae**(*formulae*, *amounts=None*)

Adds together a set of formulae.

Parameters

- **formulae** (*list of dictionary or counter objects*) – List of chemical formulae.
- **amounts** (*list of floats*) – List of amounts of each formula.

Returns
The sum of the user-provided formulae

Return type
Counter object

burnman.utils.chemistry.**formula_mass**(*formula*)

A function to take a chemical formula and compute the formula mass.

Parameters
formula (*dict or Counter object*) – A chemical formula

Returns
The mass per mole of formula [kg]

Return type
float

burnman.utils.chemistry.**convert_formula**(*formula*, *to_type='mass'*, *normalize=False*)

Converts a chemical formula from one type (mass or molar) into the other. Renormalises amounts if normalize=True.

Parameters

- **formula** (*dict or Counter object*) – A chemical formula.
- **to_type** (*str*) – Conversion type, one of 'mass' or 'molar'.
- **normalize** (*bool*) – Whether or not to normalize the converted formula to 1.

Returns
The converted formula.

Return type
dict

burnman.utils.chemistry.**process_solution_chemistry**(*solution_model*)

This function parses a class instance with a "site_formulae" attribute containing site information, e.g.

['[Mg]3[Al]2Si3O12', '[Mg]3[Mg1/2Si1/2]2Si3O12']

It outputs the bulk composition of each endmember (removing the site information), and also a set of variables and arrays which contain the site information. These are output in a format that can easily be used to calculate activities and gibbs free energies, given molar fractions of the phases and pressure and temperature where necessary.

Parameters

solution_model – Class must have a "site_formulae" attribute, containing a list of chemical formulae with site information

Return type

None

Note

Nothing is returned from this function, but the solution_model object gains the following attributes:

- **empty_formula [string]**
  Abbreviated chemical formula with sites denoted by empty square brackets.
- **general_formula [string]**
  General chemical formula with sites denoted by square brackets filled with a comma-separated list of species
- **n_sites [integer]**
  Number of sites in the solution. Should be the same for all endmembers.
- **sites [list of lists of strings]**
  A list of species for each site in the solution.
- **site_names [list of strings]**
  A list of species_site pairs in the solution, where each distinct site is given by a unique uppercase letter e.g. ['Mg_A', 'Fe_A', 'Al_A', 'Al_B', 'Si_B'].
- **n_occupancies [integer]**
  Sum of the number of possible species on each of the sites in the solution. Example: A binary solution [[A][B],[B][C1/2D1/2]] would have n_occupancies = 5, with two possible species on Site 1 and three on Site 2.
- **site_multiplicities [2D array of floats]**
  A 1D array for each endmember in the solution, containing the multiplicities of each site per formula unit. To simplify computations later, the multiplicities are repeated for each species on each site, so the shape of this attribute is (n_endmembers, n_site_species).
- **endmember_occupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the fraction of atoms of each species on each site.
- **endmember_noccupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the number of atoms of each species on each site per mole of endmember.

burnman.utils.chemistry.**site_occupancies_to_strings**(*site_species_names*, *site_multiplicities*, *endmember_occupancies*)

Converts a list of endmember site occupancies into a list of string representations of those occupancies.

Parameters

- **site_species_names** (*2D list of strings*) – A list of list of strings, giving the names of the species which reside on each site. List of sites, each of which contains a list of the species occupying each site.
- **site_multiplicities** (*1D or 2D numpy array of floats*) – List of floats giving the multiplicity of each site. If 2D, must have the same shape as endmember_occupancies. If 1D, must be either the same length as the number of sites, or the same length as site_species_names (with an implied repetition of the same number for each species on a given site).
- **endmember_occupancies** (*2D numpy array of floats*) – A list of site-species occupancies for each endmember. The first dimension loops over the endmembers, and the second dimension loops over the site-species occupancies for that endmember. The total number and order of occupancies must be the same as the strings in site_species_names.

Returns
A list of strings in standard burnman format. For example, [Mg]3[Al]2 would correspond to the classic two-site pyrope garnet.

Return type
list of strings

burnman.utils.chemistry.**compositional_array**(*formulae*)

Parameters
formulae (*list of dicts*) – List of chemical formulae

Returns
Array of endmember formulae and a list of elements.

Return type
2D numpy.array of floats and a list of strs

burnman.utils.chemistry.**ordered_compositional_array**(*formulae*, *elements*)

Parameters
formulae (*list of dicts*) – List of chemical formulae

:param elements : List of elements :type elements: list of strings

Returns
Array of endmember formulae

Return type
2D array of floats

burnman.utils.chemistry.**formula_to_string**(*formula*, *use_fractions=True*, *significant_figures=2*, *latex=False*)

Parameters

- **formula** (*dict or Counter*) – Chemical formula

- **use_fractions** (*bool*) – Whether or not to use fractions in the output string. If False, numbers are rounded to the number of significant figures given by the significant_figures parameter.
- **significant_figures** (*int*) – Number of significant figures to use if use_fractions=False.
- **latex** (*bool*) – Whether or not to format the output string for use in LaTeX.

Returns
A formula string, with element order as given in the list IUPAC_element_order. If one or more keys in the dictionary are not one of the elements in the periodic table, then they are added at the end of the string.

Return type
str

burnman.utils.chemistry.**sort_element_list_to_IUPAC_order**(*element_list*)

:param element_list : List of elements. :type element_list: list

Returns
List of elements sorted into IUPAC order

Return type
list

burnman.utils.chemistry.**convert_fractions**(*composite*, *phase_fractions*, *input_type*, *output_type*)

Takes a composite with a set of user defined molar, volume or mass fractions (which do not have to be the fractions currently associated with the composite) and converts the fractions to molar, mass or volume.

Conversions to and from mass require a molar mass to be defined for all phases. Conversions to and from volume require set_state to have been called for the composite.

Parameters

- **composite** (*Composite*) – Composite for which fractions are to be defined.
- **phase_fractions** (*list of floats*) – List of input phase fractions (of type input_type).
- **input_type** (*str*) – Input fraction type. One of 'molar', 'mass' or 'volume'.
- **output_type** (*str*) – Output fraction type. One of 'molar', 'mass' or 'volume'.

Returns
List of output phase fractions (of type output_type)

Return type
list of floats

burnman.utils.chemistry.**reaction_matrix_as_strings**(*reaction_matrix*, *compound_names*)

Returns a list of string representations of all the reactions in reaction_matrix.

Parameters

- **reaction_matrix** (*2D numpy array*) – Matrix of stoichiometric amounts of each compound j in reaction i.
- **compound_names** (*list of strings*) – List of compound names.

Returns
List of strings corresponding to each reaction.

Return type
list of strings

burnman.tools.chemistry.**fugacity**(*standard_material*, *assemblage*)

Calculates the fugacity of a standard material in another assemblage.

> **Note**
> set_method and set_state should already have been used on both assemblages.

Parameters

- **standard_material** – Standard material for which to calculate the fugacity. The material must have a formula as a dictionary parameter.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.

Returns
Value of the fugacity of the component with respect to the standard material.

Return type
float

burnman.tools.chemistry.**relative_fugacity**(*component_formula*, *assemblage*, *reference_assemblage*)

Calculates the fugacity of a chemical component in one assemblage relative to another one.

> **Note**
> set_method and set_state should already have been used on both assemblages.

Parameters

- **component_formula** (*dictionary*) – Chemical formula for which to compute the relative fugacity.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.
- **reference_assemblage** (*burnman.Composite*) – Reference assemblage against which to measure the fugacity.

Returns
Value of the fugacity of the component in the assemblage with respect to the reference_assemblage.

Return type
float

burnman.tools.chemistry.**equilibrium_pressure**(*minerals*, *stoichiometry*, *temperature*, *pressure_initial_guess=100000.0*)

Given a list of minerals, their reaction stoichiometries and a temperature of interest, compute the equilibrium pressure of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **temperature** (*float*) – Temperature of interest [K].
- **pressure_initial_guess** (*float*) – Initial pressure guess [Pa].

Returns
The equilibrium pressure of the reaction [Pa].

Return type
float

burnman.tools.chemistry.**equilibrium_temperature**(*minerals*, *stoichiometry*, *pressure*, *temperature_initial_guess=1000.0*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**invariant_point**(*minerals_r1*, *stoichiometry_r1*, *minerals_r2*, *stoichiometry_r2*, *pressure_temperature_initial_guess=[1000000000.0, 1000.0]*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**hugoniot**(*mineral*, *P_ref*, *T_ref*, *pressures*, *reference_mineral=None*)

Calculates the temperatures (and volumes) along a Hugoniot as a function of pressure according to the Hugoniot equation U2-U1 = 0.5*(p2 - p1)(V1 - V2) where U and V are the internal energies and volumes (mass or molar) and U = F + TS

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the Hugoniot is to be calculated.
- **P_ref** (*float*) – Reference pressure [Pa]
- **T_ref** (*float*) – Reference temperature [K]
- **pressures** (*numpy.array of floats*) – Set of pressures [Pa] for which the Hugoniot temperature and volume should be calculated.
- **reference_mineral** (*burnman.Mineral*) – Mineral which is stable at the reference conditions Provides an alternative U_0 and V_0 when the reference mineral transforms to the mineral of interest at some (unspecified) pressure.

Returns
The Hugoniot temperatures and volumes at the given pressures.

Return type
tuple of numpy.arrays

burnman.tools.chemistry.**reactions_from_stoichiometric_matrix**(*stoichiometric_matrix*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters
stoichiometric_matrix (*2D numpy array*) – An array of the stoichiometric (molar) amounts of component j in compound i.

Returns
An array of the stoichiometric (molar) amounts of compound j in reaction i.

Return type
2D numpy array

burnman.tools.chemistry.**reactions_from_formulae**(*formulae*, *compound_names*, *return_strings=True*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters

- **formulae** (*list of dictionaries or list of strings*) – List of the chemical formulae, either as strings or as a list of dictionaries of elements.
- **compound_names** (*list of strings*) – List of the compound names in the formula list.
- **return_strings** (*bool*) – Whether to return the reactions as strings or array.

Returns
Either a 2D array of the stoichiometric (molar) amounts of compound j in reaction i, or a list of strings. The parameter compound_names is only used if strings are requested.

Return type
2D numpy array or list of strings

20.18 Equilibrium Thermodynamics

burnman.tools.equilibration.**equilibrate**(*composition*, *assemblage*, *equality_constraints*, *free_compositional_vectors=[]*, *tol=None*, *store_iterates=False*, *store_assemblage=True*, *max_iterations=100.0*, *verbose=False*)

A function that finds the thermodynamic equilibrium state of an assemblage subject to given equality constraints by solving a set of nonlinear equations related to the chemical potentials and other state variables of the system.

The user chooses an assemblage (e.g. olivine, garnet and orthopyroxene) and $2 + n_c$ equality constraints, where n_c is the number of bulk compositional degrees of freedom. The equilibrate function attempts to find the remaining unknowns that satisfy those constraints.

There are a number of equality constraints implemented in burnman. These are given as a list of lists. Each constraint should have the form: [<constraint type>, <constraint>], where <constraint type> is one of 'P', 'T', 'S', 'V', 'X', 'phase_fraction', or 'phase_composition'. The format of the <constraint> object depends on the constraint type:

- **P: float or numpy.array of**
  pressures [Pa]
- **T: float or numpy.array of**
  temperatures [K]

- **S: float or numpy.array of**
  entropies [J/K]
- **V: float or numpy.array of**
  volumes [m:math:^3]
- **phase_fraction: tuple with the form (<phase> <fraction(s)>),**
  where <phase> is one of the phase objects in the assemblage and <fraction(s)> is a float or numpy.array corresponding to the desired phase fractions.
- **phase_composition: tuple with the form (<phase> <constraint>),**
  where <phase> is one of the phase objects in the assemblage and <constraint> has the form (site_names, n, d, v), where $(nx)/(dx) = v$, n and d are constant vectors of site coefficients, and v is a float or numpy.array. For example, a constraint of the form ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) would correspond to equal amounts Mg and Fe on the A site (Mg_A / (Mg_A + Fe_A) = 0.5).
- **X: list of a numpy.array and a float or numpy.array,**
  where the equality satisfies the linear equation arr[0] x = arr[1]. The first numpy.array is fixed, and the second is to be looped over by the equilibrate function. This is a generic compositional equality constraint.

Parameters

- **composition** (*dict*) – The bulk composition that the assemblage must satisfy.
- **assemblage** (*burnman.Composite* or *burnman.RelaxedComposite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of list*) – The list of equality constraints. See above for valid formats.
- **free_compositional_vectors** (*list of dict*) – A list of dictionaries containing the compositional freedom of the solution. For example, if the list contains the vector {'Mg': 1., 'Fe': -1}, that implies that the bulk composition is equal to composition + a (n_Mg - n_Fe), where the value of a is to be determined by the solve. Vector given in atomic (molar) units of elements.
- **tol** (*float or numpy.array*) – The tolerance for the nonlinear solver. This can be a single float, which is then applied to all parameters, or a numpy array with the same length as the number of parameters in the solve (2 + number of phases + number of free compositional vectors). The default is None, which sets the tolerances to 1 Pa for pressure, 1e-3 K for temperature, 1e-6 for phase amounts and 1e-6 for free compositional vectors. One of the termination criteria for the nonlinear solver is that the absolute change in each parameter is less than the corresponding tolerance.
- **store_iterates** (*bool*) – Whether to store the parameter values for each iteration in each solution object.
- **store_assemblage** (*bool*) – Whether to store a copy of the assemblage object in each solution object.

- **max_iterations** (*int*) – The maximum number of iterations for the nonlinear solver.
- **verbose** (*bool*) – Whether to print output updating the user on the status of equilibration.

Returns

Solver solution object (or a list, or a 2D list of solution objects) created by *burnman.optimize.nonlinear_solvers.damped_newton_solve()*, and a namedtuple object created by *burnman.tools.equilibration.get_equilibration_parameters()*. See documentation of these functions for the return types. If store_assemblage is True, each solution object also has an attribute called *assemblage*, which contains a copy of the input assemblage with the equilibrated properties. So, for a 2D grid of solution objects, one could call either sols[0][1].x[0] or sols[0][1].assemblage.pressure to get the pressure.

Return type

tuple

burnman.tools.equilibration.**get_equilibration_parameters**(*assemblage*, *composition*, *free_compositional_vectors*)

Builds a named tuple containing the parameter names and various other parameters needed by the equilibrium solve.

Parameters

- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **composition** (*dict*) – The bulk composition for the equilibrium problem.
- **free_compositional_vectors** (*list of dictionaries*) – The bulk compositional degrees of freedom for the equilibrium problem.

Returns

A tuple with attributes:

- **parameter_names: list of strings, the names of the parameters** for the current equilibrium problem
- default_tolerances: numpy.array, the default tolerances for each parameter
- n_parameters: int, the number of parameters for the current equilibrium problem
- **phase_amount_indices: list of int, the indices of the parameters that** correspond to phase amounts
- bulk_composition_vector: numpy.array, the bulk composition vector
- free_compositional_vectors: 2D numpy.array, the free compositional vectors
- **reduced_composition_vector: numpy.array, the bulk composition vector** reduced to the independent elements
- **reduced_free_composition_vectors: 2D numpy.array, the free compositional vectors** reduced to the independent elements

- **constraint_vector: numpy.array, vector for the linear constraints bounding the**
  valid parameter space (b in Ax + b < eps)
- **constraint_matrix: 2D numpy.array, matrix for the linear constraints bounding the**
  valid parameter space (A in Ax + b < eps)

Return type
namedtuple

20.19 Seismic Models

- *Base class for all seismic models*
- *Class for 1D Models*
- *Models currently implemented*
- *Attenuation Correction*

20.19.1 Base class for all seismic models

class burnman.classes.seismic.**Seismic1DModel**

Bases: object

Base class for all the seismological models.

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

internal_depth_list(*mindepth=0.0*, *maxdepth=1e+99*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`density`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

`G`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`K`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

`QK`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

20.19.2 Class for 1D Models

class burnman.classes.seismic.**SeismicTable**

Bases: *Seismic1DModel*

This is a base class that gets a 1D seismic model from a table indexed and sorted by radius. Fill the tables in the constructor after deriving from this class. This class uses `burnman.seismic.Seismic1DModel`

Note: all tables need to be sorted by increasing depth. self.table_depth needs to be defined. Alternatively, you can also overwrite the _lookup function if you want to access with something else.

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].

- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

radius(*pressure*)

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

20.19.3 Models currently implemented

class burnman.classes.seismic.**PREM**

Bases: *SeismicTable*

Reads PREM (1s) (input_seismic/prem.txt, [DA81]). See also burnman.seismic.SeismicTable.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].

- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**Slow**

Bases: *SeismicTable*

Inserts the mean profiles for slower regions in the lower mantle (Lekic et al. 2012). We stitch together tables 'input_seismic/prem_lowermantle.txt', 'input_seismic/swave_slow.txt', 'input_seismic/pwave_slow.txt'). See also `burnman.seismic.SeismicTable`.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`density`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

`depth`(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`evaluate`(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

`gravity`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns

Depths [m].

Return type

numpy.array

pressure(*depth*)

Parameters

depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns

Pressure(s) at given depth(s) in [Pa].

Return type

float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters

depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns

P wave velocity at given depth(s) in [m/s].

Return type

float or numpy.array of floats

v_phi(*depth*)

Parameters

depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns

Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**Fast**

Bases: *SeismicTable*

Inserts the mean profiles for faster regions in the lower mantle (Lekic et al. 2012). We stitch together tables 'input_seismic/prem_lowermantle.txt', 'input_seismic/swave_fast.txt', 'input_seismic/pwave_fast.txt'). See also burnman.seismic.Seismic1DModel.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

`QK`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

`bullen`(*depth*)

Returns the Bullen parameter only for significant arrays

`density`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

`depth`(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`evaluate`(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_phi`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`v_s`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

`class burnman.classes.seismic.STW105`

Bases: *SeismicTable*

Reads STW05 (a.k.a. REF) (1s) (input_seismic/STW105.txt, [KED08]). See also `burnman.seismic.SeismicTable`.

`G`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

`K`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.

- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**IASP91**

Bases: *SeismicTable*

Reads REF/STW05 (input_seismic/STW105.txt, [KED08]). See also `burnman.seismic.SeismicTable`.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

`evaluate`(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

`gravity`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

`internal_depth_list`(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].
- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

`pressure`(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

class burnman.classes.seismic.**AK135**

Bases: *SeismicTable*

Reads AK135 (input_seismic/ak135.txt, [KEB95]). See also `burnman.seismic.SeismicTable`.

G(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Shear modulus at given depth(s) in [Pa].

Return type
float or numpy.array of floats

K(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk modulus at given depth(s) in [Pa]

Return type
float or numpy.array of floats

QG(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for shear modulus at given depth(s).

Return type
float or numpy.array of floats

QK(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Quality factor (dimensionless) for bulk modulus at given depth(s).

Return type
float or numpy.array of floats

bullen(*depth*)

Returns the Bullen parameter only for significant arrays

density(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Density at given depth(s) in [kg/m^3].

Return type
float or numpy.array of floats

depth(*pressure*)

Parameters
pressure (*float or numpy.array of floats*) – Pressure(s) [Pa] at which to evaluate depths.

Returns
Depth(s) [m] for given pressure(s).

evaluate(*vars_list*, *depth_list=None*)

Returns the lists of data for a Seismic1DModel for the depths provided

Parameters

- **vars_list** (*array of str*) – Available variables depend on the seismic model, and can be chosen from 'pressure', 'density', 'gravity', 'v_s', 'v_p', 'v_phi', 'G', 'K', 'QG' and 'QK'.
- **depth_list** (*array of floats*) – Array of depths [m] to evaluate seismic model at.

Returns
Array of values shapes as (len(vars_list),len(depth_list)).

Return type
numpy.array

gravity(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Gravity at given depths in [m/s^2].

Return type
float or numpy.array of floats

internal_depth_list(*mindepth=0.0*, *maxdepth=10000000000.0*, *discontinuity_interval=1.0*)

Returns a sorted list of depths at which this seismic data is specified. This allows you to compare the seismic data without interpolation. The depths can be bounded by the mindepth and maxdepth parameters.

Parameters

- **mindepth** (*float*) – Minimum depth value to be returned [m].
- **maxdepth** (*float*) – Maximum depth value to be returned [m].

- **discontinuity_interval** (*float*) – Shift continuities to remove ambigious values for depth [m].

Returns
Depths [m].

Return type
numpy.array

pressure(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Pressure(s) at given depth(s) in [Pa].

Return type
float or numpy.array of floats

radius(*pressure*)

v_p(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
P wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_phi(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
Bulk sound wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

v_s(*depth*)

Parameters
depth (*float or numpy.array of floats*) – Depth(s) [m] at which to evaluate seismic model.

Returns
S wave velocity at given depth(s) in [m/s].

Return type
float or numpy.array of floats

20.19.4 Attenuation Correction

burnman.tools.seismic.**attenuation_correction**(*v_p*, *v_s*, *v_phi*, *Qs*, *Qphi*)

Applies the attenuation correction following [MBR+07], 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.tools.seismic.attenuation_correction()*

Parameters

- **v_p** (*float*) – P wave velocity in [m/s].
- **v_s** (*float*) – S wave velocitiy in [m/s].
- **v_phi** (*float*) – Bulk sound velocity in [m/s].
- **Qs** (*float*) – Shear quality factor [dimensionless].
- **Qphi** (*float*) – Bulk quality factor [dimensionless].

Returns
Corrected P wave, S wave and bulk sound velocities in [m/s].

Return type
tuple

20.20 Mineral databases

This package contains thermodynamic models from a variety of published sources:

- *DKS_2013_liquids*
- *DKS_2013_solids*
- *HGP_2018_ds633*
- *HHPH_2013*
- *HP_2011_ds62*
- *HP_2011_fluids*
- *Irving et al. 2018*
- *JH_2015*
- *Matas_etal_2007*
- *Murakami_2013*

- *Murakami_etal_2012*
- *RS_2014_liquids*
- *Saxena and Eriksson (2015)*
- *SLB_2005*
- *SLB_2011*
- *SLB_2011_ZSB_2013*
- *SLB_2024*
- *ig50NCKFMASHTOCr*
- *ig50NCKFMASTOCr*
- *mb50NCKFMASHTO*
- *mp50KFMASH*
- *mp50MnNCKFMASHTO*
- *mp50NCKFMASHTO*
- *Other minerals*

20.20.1 DKS_2013_liquids

Liquids from de Koker and Stixrude (2013) FPMD simulations.

burnman.minerals.DKS_2013_liquids.**vector_to_array**(*a*, *Of*, *Otheta*)

class burnman.minerals.DKS_2013_liquids.**SiO2_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSiO3_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi2O5_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi3O7_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgSi5O11_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg2SiO4_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg3Si2O7_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**Mg5Si07_liquid**

Bases: *Mineral*

class burnman.minerals.DKS_2013_liquids.**MgO_liquid**

Bases: *Mineral*

20.20.2 DKS_2013_solids

Solids from de Koker and Stixrude (2013) FPMD simulations

class burnman.minerals.DKS_2013_solids.**stishovite**

Bases: *Mineral*

class burnman.minerals.DKS_2013_solids.**perovskite**

Bases: *Mineral*

class burnman.minerals.DKS_2013_solids.**periclase**

Bases: *Mineral*

20.20.3 HGP_2018_ds633

Endmember minerals and melt solutions from Holland, Green and Powell (2018) and references therein. Dataset version 6.33. The values in this document are all in S.I. units, unlike those in the original tc-ds633.txt. The endmember section of this file is autogenerated using HGP633data_to_burnman.py.

class burnman.minerals.HGP_2018_ds633.**fo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**teph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lrn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mont**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chum**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chdr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mwd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fwd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mrw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**frw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**apv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**npv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ppv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cpv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**maj**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nagt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**py**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**alm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spss**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**andr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ski**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**knor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**uv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osma**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osmm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**osfa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**vsv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**andalusite**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ky**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sill**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**smul**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**amul**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tpz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnctd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**merw**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ep**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fep**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pmt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**law**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mpm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**jgd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**geh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ak**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rnk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ty**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**crd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hcrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mncrd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phA**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phD**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phE**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**shB**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cstn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zrc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**zrt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tcn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**en**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pren**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cen**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hen**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hfs**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fs**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mgts**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**di**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hed**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**jd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kjd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**acm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kos**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cats**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**caes**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rhod**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pxmn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pswo**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fact**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ts**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**parg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fgl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nyb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rieb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fanth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cumm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**grun**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ged**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spr4**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**spr5**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fspr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mcar**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcar**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**deer**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cel**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fcel**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ma**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**phl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ann**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnbi**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**east**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**naph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tan**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**clin**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ames**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**afchl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**daph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mnchl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sud**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fsud**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**prl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tats**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tap**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nta**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**minn**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**minm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kao**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pre**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fpre**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**chr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**liz**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**glt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fstp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mstp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**atg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ab**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**abh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mic**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**san**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**an**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kcm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wa**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hol**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**q**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**crst**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**coe**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**stv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ne**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cg**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cgh**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**macf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mscf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fscf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nacf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cacf**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**manal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**nanal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**msnal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fsnal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**canal**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sdl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kls**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**me**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wrk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lmt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**heu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**stlb**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anl**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lime**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ru**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**per**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**fper**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**wu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mang**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mcor**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hem**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**esk**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bix**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**NiO**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pnt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**geik**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ilm**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdy**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyT**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyC**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ten**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cup**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**herc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mft**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**qnd**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**usp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**picr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**br**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**dsp**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gth**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**cc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**arag**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**mag**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**sid**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**rhc**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**dol**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ank**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**syv**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hlt**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**pyr**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trot**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**tro**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lot**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**trov**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**any**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**iron**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**Ni**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**Cu**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**gph**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**diam**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**S**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**syvL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hltL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**perL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**limL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**corL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**eskL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**hemL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**qL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**h2oL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**foL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**faL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**woL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**enL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**diL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**silL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**anL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**kspL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**abL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**neL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**lcL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**ruL**

Bases: *Mineral*

class burnman.minerals.HGP_2018_ds633.**bdyL**

Bases: *Mineral*

burnman.minerals.HGP_2018_ds633.**cov**()

A function which loads and returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset.

Returns

covariance matrix dictionary, with keys 'endmember_names' and 'covariance_matrix'. Endmember names are in the same order as the rows and columns of the covariance matrix.

Return type

dict

burnman.minerals.HGP_2018_ds633.**make_melt_class**(*selected_endmembers*)

A function that generates a melt class that is a subclass of the Holland et al. (2018) silicate melt model.

Parameters

selected_endmembers: list of Minerals

Endmembers to include in the model. Valid endmembers are given in the following list, and should be specified in the same order as they appear in the list: ['q4L', 'sl1L', 'wo1L', 'fo2L', 'fa2L', 'jdL', 'hmL', 'ekL', 'tiL', 'kjL', 'ctL', 'h2o1L'].

Returns

melt_class: Solution class

Melt class spanning the specified endmembers.

class burnman.minerals.HGP_2018_ds633.**silicate_melt**(*molar_fractions=None*)

Bases: *Solution*

burnman.minerals.HGP_2018_ds633.**CMS_melt**

alias of silicate_melt

burnman.minerals.HGP_2018_ds633.**MS_melt**

alias of silicate_melt

HGP_2018 (ds-633) zero-point energy covariance matrix Derived 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.

20.20.4 HHPH_2013

Minerals from Holland et al. (2013) and references therein. The values in this document are all in S.I. units, unlike those in the original paper. File autogenerated using HHPHdata_to_burnman.py.

class burnman.minerals.HHPH_2013.**fo**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fa**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mwd**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fwd**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mrw**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**frw**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**apv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**npv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cpv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mak**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fak**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**maj**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**nagt**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**py**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**alm**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**gr**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**en**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cen**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hen**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hfs**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fs**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mgts**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**di**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**hed**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**jd**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cats**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**stv**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**macf**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mscf**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fscf**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**nacf**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cacf**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**manal**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**nanal**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**msnal**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fsnal**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**canal**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**per**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**fper**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**cor**

Bases: *Mineral*

class burnman.minerals.HHPH_2013.**mcor**

Bases: *Mineral*

20.20.5 HP_2011_ds62

Endmember minerals from Holland and Powell 2011 and references therein. Update to dataset version 6.2. The values in this document are all in S.I. units, unlike those in the original tc-ds62.txt. File autogenerated using HPdata_to_burnman.py.

class burnman.minerals.HP_2011_ds62.**fo**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fa

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.teph

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.lrn

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mont

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.chum

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.chdr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mwd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fwd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mrw

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.frw

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.apv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cpv

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mak

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fak

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.maj

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**py**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**alm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spss**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**gr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**andr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**knor**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osma**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osmm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**osfa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**vsv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**andalusite**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ky**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sill**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**smul**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**amul**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tpz**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnctd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**merw**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**zo**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cz**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ep**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fep**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pmt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**law**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mpm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fpm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**jgd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**geh**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ak**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**rnk**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ty**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**crd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hcrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mncrd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**phA**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cstn**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**zrc**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**en**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pren**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cen**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hen**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fs**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mgts**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.di

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.hed

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.jd

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.acm

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.kos

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cats

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.caes

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.rhod

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pxmn

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.wo

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pswo

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.wal

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.tr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fact

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ts

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.parg

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.gl

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fgl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**rieb**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**anth**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fanth**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cumm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**grun**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ged**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spr4**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**spr5**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fspr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mcar**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcar**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**deer**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cel**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fcel**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ma**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**phl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ann**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnbi**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**east**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**naph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**clin**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ames**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**afchl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**daph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**mnchl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sud**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fsud**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**prl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ta**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**fta**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tats**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.tap
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.minn
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.minm
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.kao
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pre
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fpre
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.chr
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.liz
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.glt
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fstp
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mstp
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.atg
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ab
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.abh
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mic
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.san
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.an
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**kcm**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**wa**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hol**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**q**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trd**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**crst**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**coe**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**stv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ne**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cg**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**cgh**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**sdl**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**kls**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lc**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**me**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**wrk**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lmt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.heu
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.stlb
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.anl
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.lime
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ru
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.per
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.fper
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mang
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cor
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mcor
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.hem
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.esk
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.bix
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.NiO
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.pnt
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.geik
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ilm
Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.bdy

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.ten

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cup

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.sp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.herc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mt

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mft

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.usp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.picr

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.br

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.dsp

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.gth

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.cc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.arag

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.mag

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.sid

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.rhc

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**dol**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**ank**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**syv**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**hlt**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**pyr**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trot**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**tro**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**lot**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**trov**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**any**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**iron**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**Ni**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**Cu**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**gph**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**diam**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**S**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.**syvL**

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.hltL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.perL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.limL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.corL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.qL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.h2oL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.foL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.faL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.woL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.enL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.diL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.silL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.anL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.kspL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.abL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.neL

Bases: *Mineral*

class burnman.minerals.HP_2011_ds62.lcL

Bases: *Mineral*

burnman.minerals.HP_2011_ds62.**cov**()

A function which loads and returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset.

Returns

covariance matrix dictionary, with keys 'endmember_names' and 'covariance_matrix'. Endmember names are in the same order as the rows and columns of the covariance matrix.

Return type

dict

HP_2011 (ds-62) zero-point energy covariance matrix Derived 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

20.20.6 HP_2011_fluids

Fluids from Holland and Powell 2011 and references therein. CORK parameters are taken from various sources.

CHO gases from Holland and Powell, 1991:

- ["CO2",304.2,0.0738]
- ["CH4",190.6,0.0460]
- ["H2",41.2,0.0211]
- ["CO",132.9,0.0350]

H2O and S2 from Wikipedia, 2012/10/23:

- ["H2O",647.096,0.22060]
- ["S2",1314.00,0.21000]

H2S from ancyclopedia.airliquide.com, 2012/10/23:

- ["H2S",373.15,0.08937]

NB: Units for cork[i] in Holland and Powell datasets are:

- a = kJ^2/kbar*K^(1/2)/mol^2: multiply by 1e-2
- b = kJ/kbar/mol: multiply by 1e-5
- c = kJ/kbar^1.5/mol: multiply by 1e-9
- d = kJ/kbar^2/mol: multiply by 1e-13

Individual terms are divided through by P, P, P^1.5, P^2, so:

- [0][j]: multiply by 1e6
- [1][j]: multiply by 1e3
- [2][j]: multiply by 1e3

- [3][j]: multiply by 1e3
- cork_P is given in kbar: multiply by 1e8

class burnman.minerals.HP_2011_fluids.**CO2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**CH4**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**O2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**H2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**S2**

Bases: *Mineral*

class burnman.minerals.HP_2011_fluids.**H2S**

Bases: *Mineral*

20.20.7 Irving et al. 2018

Seismically derived EoS parameters for the outer core using normal mode center frequencies. These parameters are derived parameterizing the outer core velocity and density with an equation-of-state, and inverting for K, Kprime, and molar_volume (molar_mass is fixed to 0.05, as sensitivy of the normal modes is only to molar density). The equations-of-state are assumed to capture the isentropic behaviour across the outer core. Potentially, these parameters can be used to represent cores of other planets. See: Irving, Cottaar, Lekic (2018) Seismically determined elastic parameters for the outer core, Science Advances

class burnman.minerals.ICL_2018.**EPOC_vinet**

Bases: *Mineral*

class burnman.minerals.ICL_2018.**EPOC_bm**

Bases: *Mineral*

20.20.8 JH_2015

Solid solutions from Jennings and Holland, 2015 and references therein (10.1093/petrology/egv020). The values in this document are all in S.I. units, unlike those in the original tc file.

class burnman.minerals.JH_2015.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**cfs**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**crdi**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**cess**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**cen**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**cfm**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.JH_2015.**fm**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**odi**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**cren**

Bases: *CombinedMineral*

class burnman.minerals.JH_2015.**mess**

Bases: *CombinedMineral*

burnman.minerals.JH_2015.**construct_combined_covariance**(*original_covariance_dictionary*, *combined_mineral_list*)

This function takes a dictionary containing a list of endmember_names and a covariance_matrix, and a list of CombinedMineral instances, and creates an updated covariance dictionary containing those CombinedMinerals

Parameters

original_covariance_dictionary

[dictionary] Contains a list of strings of endmember_names of length n and a 2D numpy array covariance_matrix of shape n x n

combined_mineral_list
[list of instances of *burnman.CombinedMineral*] List of minerals to be added to the covariance matrix

Returns

cov
[dictionary] Updated covariance dictionary, with the same keys as the original

burnman.minerals.JH_2015.**cov**()

A function which returns the variance-covariance matrix of the zero-point energies of all the endmembers in the dataset. Derived from HP_2011_ds62, modified to include all the new CombinedMinerals.

Returns

cov
[dictionary] Dictionary keys are: - endmember_names: a list of endmember names, and - covariance_matrix: a 2D variance-covariance array for the endmember enthalpies of formation

from Kurnosov et al. Nature 2017 Kurnosov, A., et al. "Evidence for a Fe3+-rich pyrolitic lower mantle from (Al, Fe)-bearing bridgmanite elasticity data." Nature 543.7646 (2017): 543-546. doi:10.1038/nature21390

class burnman.minerals.KMFBZ_2017.**bridgmanite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.KMFBZ_2017.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.KMFBZ_2017.**mg_si_perovskite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**fe_si_perovskite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**fe_al_perovskite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**al_al_perovskite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**periclase**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**wuestite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**ca_perovskite**

Bases: *Mineral*

class burnman.minerals.KMFBZ_2017.**hcp_iron**

Bases: *Mineral*

burnman.minerals.KMFBZ_2017.**perovskite**

alias of bridgmanite

20.20.9 Matas_etal_2007

Minerals from Matas et al. 2007 and references therein. See Table 1 and 2.

class burnman.minerals.Matas_etal_2007.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**fe_perovskite**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**al_perovskite**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**ca_perovskite**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**periclase**

Bases: *Mineral*

class burnman.minerals.Matas_etal_2007.**wuestite**

Bases: *Mineral*

burnman.minerals.Matas_etal_2007.**ca_bridgmanite**

alias of ca_perovskite

burnman.minerals.Matas_etal_2007.**mg_bridgmanite**

alias of mg_perovskite

burnman.minerals.Matas_etal_2007.**fe_bridgmanite**

alias of fe_perovskite

burnman.minerals.Matas_etal_2007.**al_bridgmanite**

alias of al_perovskite

20.20.10 Murakami_2013

Minerals from Murakami 2013 and references therein.

class burnman.minerals.Murakami_2013.**periclase**

Bases: *Mineral*

class burnman.minerals.Murakami_2013.**wuestite**

Bases: *Mineral*

Murakami 2013 and references therein

class burnman.minerals.Murakami_2013.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_2013.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.Murakami_2013.**mg_bridgmanite**

alias of mg_perovskite

burnman.minerals.Murakami_2013.**fe_bridgmanite**

alias of fe_perovskite

20.20.11 Murakami_etal_2012

Minerals from Murakami et al. (2012) supplementary table 5 and references therein, V_0 from Stixrude & Lithgow-Bertolloni 2005. Some information from personal communication with Murakami.

class burnman.minerals.Murakami_etal_2012.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_perovskite_3rdorder**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_perovskite**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**mg_periclase**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase**

Bases: *HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_3rd**

Bases: *HelperSpinTransition*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_HS_3rd**

Bases: *Mineral*

class burnman.minerals.Murakami_etal_2012.**fe_periclase_LS_3rd**

Bases: *Mineral*

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite**

alias of mg_perovskite

burnman.minerals.Murakami_etal_2012.**fe_bridgmanite**

alias of fe_perovskite

burnman.minerals.Murakami_etal_2012.**mg_bridgmanite_3rdorder**

alias of mg_perovskite_3rdorder

20.20.12 RS_2014_liquids

Liquids from Ramo and Stixrude (2014) FPMD simulations. There are some typos in the article which have been corrected where marked with the help of David Munoz Ramo.

class burnman.minerals.RS_2014_liquids.**Fe2SiO4_liquid**

Bases: *Mineral*

20.20.13 Saxena and Eriksson (2015)

Iron endmember minerals and melt taken from [SE15] using the equation of state of [BMS07].

1 bar gibbs free energy coefficients are given in the following order: [[T_max, [const, T, T*ln(T), T^(-1), T^(-2), T^(-3), T^(-9), T^2, T^3, T^4, T^7, T^(1/2), ln(T)]]

class burnman.minerals.SE_2015.**bcc_iron**

Bases: *Mineral*

BCC iron from [SE15].

class burnman.minerals.SE_2015.**fcc_iron**

Bases: *Mineral*

FCC iron from [SE15].

class burnman.minerals.SE_2015.**hcp_iron**

Bases: *Mineral*

HCP iron from [SE15].

class burnman.minerals.SE_2015.**liquid_iron**

Bases: *Mineral*

Liquid iron from [SE15].

20.20.14 SLB_2005

Minerals from Stixrude & Lithgow-Bertelloni 2005 and references therein

class burnman.minerals.SLB_2005.**stishovite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2005.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.SLB_2005.**mg_bridgmanite**

alias of mg_perovskite

burnman.minerals.SLB_2005.**fe_bridgmanite**

alias of fe_perovskite

20.20.15 SLB_2011

Minerals from Stixrude & Lithgow-Bertelloni 2011 and references therein. File autogenerated using SLB-data_to_burnman.py.

class burnman.minerals.SLB_2011.**c2c_pyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**ca_ferrite_structured_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**akimotoite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**post_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_ringwoodite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_aluminous_spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**mg_fe_wadsleyite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2011.**anorthite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**albite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**spinel**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**hercynite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**forsterite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fayalite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_wadsleyite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_wadsleyite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_ringwoodite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_ringwoodite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**enstatite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**ferrosilite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_tschermaks**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**ortho_diopside**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**diopside**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**hedenbergite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**clinoenstatite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.ca_tschermaks

Bases: *Mineral*

class burnman.minerals.SLB_2011.jadeite

Bases: *Mineral*

class burnman.minerals.SLB_2011.hp_clinoenstatite

Bases: *Mineral*

class burnman.minerals.SLB_2011.hp_clinoferrosilite

Bases: *Mineral*

class burnman.minerals.SLB_2011.ca_perovskite

Bases: *Mineral*

class burnman.minerals.SLB_2011.mg_akimotoite

Bases: *Mineral*

class burnman.minerals.SLB_2011.fe_akimotoite

Bases: *Mineral*

class burnman.minerals.SLB_2011.corundum

Bases: *Mineral*

class burnman.minerals.SLB_2011.pyrope

Bases: *Mineral*

class burnman.minerals.SLB_2011.almandine

Bases: *Mineral*

class burnman.minerals.SLB_2011.grossular

Bases: *Mineral*

class burnman.minerals.SLB_2011.mg_majorite

Bases: *Mineral*

class burnman.minerals.SLB_2011.jd_majorite

Bases: *Mineral*

class burnman.minerals.SLB_2011.quartz

Bases: *Mineral*

class burnman.minerals.SLB_2011.coesite

Bases: *Mineral*

class burnman.minerals.SLB_2011.stishovite

Bases: *Mineral*

class burnman.minerals.SLB_2011.seifertite

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**al_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**al_post_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**mg_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**fe_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**na_ca_ferrite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**kyanite**

Bases: *Mineral*

class burnman.minerals.SLB_2011.**nepheline**

Bases: *Mineral*

burnman.minerals.SLB_2011.**ab**

alias of albite

burnman.minerals.SLB_2011.**an**

alias of anorthite

burnman.minerals.SLB_2011.**sp**

alias of spinel

burnman.minerals.SLB_2011.**hc**

alias of hercynite

burnman.minerals.SLB_2011.**fo**
alias of forsterite

burnman.minerals.SLB_2011.**fa**
alias of fayalite

burnman.minerals.SLB_2011.**mgwa**
alias of mg_wadsleyite

burnman.minerals.SLB_2011.**fewa**
alias of fe_wadsleyite

burnman.minerals.SLB_2011.**mgri**
alias of mg_ringwoodite

burnman.minerals.SLB_2011.**feri**
alias of fe_ringwoodite

burnman.minerals.SLB_2011.**en**
alias of enstatite

burnman.minerals.SLB_2011.**fs**
alias of ferrosilite

burnman.minerals.SLB_2011.**mgts**
alias of mg_tschermaks

burnman.minerals.SLB_2011.**odi**
alias of ortho_diopside

burnman.minerals.SLB_2011.**di**
alias of diopside

burnman.minerals.SLB_2011.**he**
alias of hedenbergite

burnman.minerals.SLB_2011.**cen**
alias of clinoenstatite

burnman.minerals.SLB_2011.**cats**
alias of ca_tschermaks

burnman.minerals.SLB_2011.**jd**
alias of jadeite

burnman.minerals.SLB_2011.**mgc2**
alias of hp_clinoenstatite

burnman.minerals.SLB_2011.**fec2**
alias of hp_clinoferrosilite

burnman.minerals.SLB_2011.**hpcen**
alias of hp_clinoenstatite

burnman.minerals.SLB_2011.**hpcfs**
alias of hp_clinoferrosilite

burnman.minerals.SLB_2011.**mgpv**
alias of mg_perovskite

burnman.minerals.SLB_2011.**mg_bridgmanite**
alias of mg_perovskite

burnman.minerals.SLB_2011.**fepv**
alias of fe_perovskite

burnman.minerals.SLB_2011.**fe_bridgmanite**
alias of fe_perovskite

burnman.minerals.SLB_2011.**alpv**
alias of al_perovskite

burnman.minerals.SLB_2011.**capv**
alias of ca_perovskite

burnman.minerals.SLB_2011.**mgil**
alias of mg_akimotoite

burnman.minerals.SLB_2011.**feil**
alias of fe_akimotoite

burnman.minerals.SLB_2011.**co**
alias of corundum

burnman.minerals.SLB_2011.**py**
alias of pyrope

burnman.minerals.SLB_2011.**al**
alias of almandine

burnman.minerals.SLB_2011.**gr**
alias of grossular

burnman.minerals.SLB_2011.**mgmj**
alias of mg_majorite

burnman.minerals.SLB_2011.**jdmj**
alias of jd_majorite

burnman.minerals.SLB_2011.**qtz**
alias of quartz

burnman.minerals.SLB_2011.**coes**
alias of coesite

burnman.minerals.SLB_2011.**st**
alias of stishovite

burnman.minerals.SLB_2011.**seif**
alias of seifertite

burnman.minerals.SLB_2011.**mppv**
alias of mg_post_perovskite

burnman.minerals.SLB_2011.**fppv**
alias of fe_post_perovskite

burnman.minerals.SLB_2011.**appv**
alias of al_post_perovskite

burnman.minerals.SLB_2011.**pe**
alias of periclase

burnman.minerals.SLB_2011.**wu**
alias of wuestite

burnman.minerals.SLB_2011.**mgcf**
alias of mg_ca_ferrite

burnman.minerals.SLB_2011.**fecf**
alias of fe_ca_ferrite

burnman.minerals.SLB_2011.**nacf**
alias of na_ca_ferrite

burnman.minerals.SLB_2011.**ky**
alias of kyanite

burnman.minerals.SLB_2011.**neph**
alias of nepheline

burnman.minerals.SLB_2011.**c2c**
alias of c2c_pyroxene

burnman.minerals.SLB_2011.**cf**
alias of ca_ferrite_structured_phase

burnman.minerals.SLB_2011.**cpx**
alias of clinopyroxene

burnman.minerals.SLB_2011.**gt**
alias of garnet

burnman.minerals.SLB_2011.**il**

alias of akimotoite

burnman.minerals.SLB_2011.**ilmenite_group**

alias of akimotoite

burnman.minerals.SLB_2011.**mw**

alias of ferropericlase

burnman.minerals.SLB_2011.**magnesiowuestite**

alias of ferropericlase

burnman.minerals.SLB_2011.**ol**

alias of mg_fe_olivine

burnman.minerals.SLB_2011.**opx**

alias of orthopyroxene

burnman.minerals.SLB_2011.**plag**

alias of plagioclase

burnman.minerals.SLB_2011.**ppv**

alias of post_perovskite

burnman.minerals.SLB_2011.**pv**

alias of mg_fe_perovskite

burnman.minerals.SLB_2011.**mg_fe_bridgmanite**

alias of mg_fe_perovskite

burnman.minerals.SLB_2011.**mg_fe_silicate_perovskite**

alias of mg_fe_perovskite

burnman.minerals.SLB_2011.**ri**

alias of mg_fe_ringwoodite

burnman.minerals.SLB_2011.**spinel_group**

alias of mg_fe_aluminous_spinel

burnman.minerals.SLB_2011.**wa**

alias of mg_fe_wadsleyite

burnman.minerals.SLB_2011.**spinelloid_III**

alias of mg_fe_wadsleyite

20.20.16 SLB_2011_ZSB_2013

Minerals from Stixrude & Lithgow-Bertelloni 2011, Zhang, Stixrude & Brodholt 2013, and references therein.

class burnman.minerals.SLB_2011_ZSB_2013.**stishovite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**periclase**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**wuestite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**mg_perovskite**

Bases: *Mineral*

class burnman.minerals.SLB_2011_ZSB_2013.**fe_perovskite**

Bases: *Mineral*

burnman.minerals.SLB_2011_ZSB_2013.**mg_bridgmanite**

alias of mg_perovskite

burnman.minerals.SLB_2011_ZSB_2013.**fe_bridgmanite**

alias of fe_perovskite

SLB_2022 Minerals from Stixrude & Lithgow-Bertelloni 2022 and references therein File autogenerated using HeFESTO_to_burnman.py and HeFESTo_parameters_010121

class burnman.minerals.SLB_2022.**ab**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**al**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**alpv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**an**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**anao**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**apbo**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**appv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**capv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**cats**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**cen**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**co**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**coes**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**crst**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**di**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**en**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fa**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fapv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fec2**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fecf**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**feil**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fepv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**feri**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fewa**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**flpv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fnal**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fo**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fppv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**fs**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**gr**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hc**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**he**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hem**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hepv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hlpv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hmag**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**hppv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**jd**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**ky**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**lppv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mag**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgc2**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgcf**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgil**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgmj**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgpv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgri**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgts**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mgwa**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mnal**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**mppv**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**nacf**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**namj**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**neph**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**nnal**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**odi**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**pe**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**py**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**qtz**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**sp**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**st**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**wu**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**wuls**

Bases: *Mineral*

class burnman.minerals.SLB_2022.**c2c_pyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**calcium_ferrite_structured_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**ilmenite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**new_aluminous_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**post_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**bridgmanite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**ringwoodite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**mg_fe_aluminous_spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2022.**wadsleyite**(*molar_fractions=None*)

Bases: *Solution*

burnman.minerals.SLB_2022.**albite**

alias of ab

burnman.minerals.SLB_2022.**almandine**
alias of al

burnman.minerals.SLB_2022.**al_perovskite**
alias of alpv

burnman.minerals.SLB_2022.**anorthite**
alias of an

burnman.minerals.SLB_2022.**alpha_nao2_phase**
alias of anao

burnman.minerals.SLB_2022.**alpha_pbo_2_sio_2**
alias of apbo

burnman.minerals.SLB_2022.**al_post_perovskite**
alias of appv

burnman.minerals.SLB_2022.**ca_perovskite**
alias of capv

burnman.minerals.SLB_2022.**lime_tschermak**
alias of cats

burnman.minerals.SLB_2022.**clinoenstatite**
alias of cen

burnman.minerals.SLB_2022.**corundum**
alias of co

burnman.minerals.SLB_2022.**coesite**
alias of coes

burnman.minerals.SLB_2022.**cristobalite**
alias of crst

burnman.minerals.SLB_2022.**diopside**
alias of di

burnman.minerals.SLB_2022.**enstatite**
alias of en

burnman.minerals.SLB_2022.**fayalite**
alias of fa

burnman.minerals.SLB_2022.**fealo3_perovskite_hs**
alias of fapv

burnman.minerals.SLB_2022.**hp_clinoferrosilite**
alias of fec2

burnman.minerals.SLB_2022.**fe_ca_ferrite**
alias of fecf

burnman.minerals.SLB_2022.**fe_akimotoite**
alias of feil

burnman.minerals.SLB_2022.**fe_perovskite**
alias of fepv

burnman.minerals.SLB_2022.**fe_ringwoodite**
alias of feri

burnman.minerals.SLB_2022.**fe_wadsleyite**
alias of fewa

burnman.minerals.SLB_2022.**fe_perovskite_low_spin**
alias of flpv

burnman.minerals.SLB_2022.**fe_nal_phase**
alias of fnal

burnman.minerals.SLB_2022.**forsterite**
alias of fo

burnman.minerals.SLB_2022.**fe_postperovskite**
alias of fppv

burnman.minerals.SLB_2022.**ferrosilite**
alias of fs

burnman.minerals.SLB_2022.**grossular**
alias of gr

burnman.minerals.SLB_2022.**hercynite**
alias of hc

burnman.minerals.SLB_2022.**hedenbergite**
alias of he

burnman.minerals.SLB_2022.**hematite**
alias of hem

burnman.minerals.SLB_2022.**fe2o3_perovskite_hs**
alias of hepv

burnman.minerals.SLB_2022.**fe2o3_perovskite_ls**
alias of hlpv

burnman.minerals.SLB_2022.**high_pressure_magnetit**
alias of hmag

burnman.minerals.SLB_2022.**hs_fe2o3_post_perovski**
alias of hppv

burnman.minerals.SLB_2022.**jadeite**
alias of jd

burnman.minerals.SLB_2022.**kyanite**
alias of ky

burnman.minerals.SLB_2022.**ls_fe2o3_post_perovski**
alias of lppv

burnman.minerals.SLB_2022.**magnetite**
alias of mag

burnman.minerals.SLB_2022.**hp_clinoenstatite**
alias of mgc2

burnman.minerals.SLB_2022.**mg_ca_ferrite**
alias of mgcf

burnman.minerals.SLB_2022.**mg_akimotoite**
alias of mgil

burnman.minerals.SLB_2022.**mg_majorite**
alias of mgmj

burnman.minerals.SLB_2022.**mg_perovskite**
alias of mgpv

burnman.minerals.SLB_2022.**mg_ringwoodite**
alias of mgri

burnman.minerals.SLB_2022.**mg_tschermaks**
alias of mgts

burnman.minerals.SLB_2022.**mg_wadsleyite**
alias of mgwa

burnman.minerals.SLB_2022.**mg_nal_phase**
alias of mnal

burnman.minerals.SLB_2022.**mg_postperovskite**
alias of mppv

burnman.minerals.SLB_2022.**na_ca_ferrite**
alias of nacf

burnman.minerals.SLB_2022.**na_majorite**
alias of namj

burnman.minerals.SLB_2022.**nepheline**
alias of neph

burnman.minerals.SLB_2022.**na_nal_phase**
alias of nnal

burnman.minerals.SLB_2022.**ortho_diopside**
alias of odi

burnman.minerals.SLB_2022.**periclase**
alias of pe

burnman.minerals.SLB_2022.**pyrope**
alias of py

burnman.minerals.SLB_2022.**quartz**
alias of qtz

burnman.minerals.SLB_2022.**spinel**
alias of sp

burnman.minerals.SLB_2022.**stishovite**
alias of st

burnman.minerals.SLB_2022.**wustite**
alias of wu

burnman.minerals.SLB_2022.**wustite_low_spin**
SOLUTION ALIASES

burnman.minerals.SLB_2022.**c2c**
alias of c2c_pyroxene

burnman.minerals.SLB_2022.**cf**
alias of calcium_ferrite_structured_phase

burnman.minerals.SLB_2022.**cpx**
alias of clinopyroxene

burnman.minerals.SLB_2022.**gt**
alias of garnet

burnman.minerals.SLB_2022.**il**
alias of ilmenite

burnman.minerals.SLB_2022.**mw**
alias of ferropericlase

burnman.minerals.SLB_2022.**nal**
alias of new_aluminous_phase

burnman.minerals.SLB_2022.**ol**
: alias of olivine

burnman.minerals.SLB_2022.**opx**
: alias of orthopyroxene

burnman.minerals.SLB_2022.**plg**
: alias of plagioclase

burnman.minerals.SLB_2022.**ppv**
: alias of post_perovskite

burnman.minerals.SLB_2022.**pv**
: alias of bridgmanite

burnman.minerals.SLB_2022.**ri**
: alias of ringwoodite

burnman.minerals.SLB_2022.**wa**
: alias of wadsleyite

20.20.17 SLB_2024

Minerals from Stixrude and Lithgow-Bertelloni (2024), and references therein.

Contains the following solutions: * c2c_pyroxene (c2c) * calcium_ferrite_structured_phase (cf) * clinopyroxene (cpx) * garnet (gt) * ilmenite (il) * ferropericlase (mw) * new_aluminous_phase (nal) * olivine (ol) * orthopyroxene (opx) * plagioclase (plg) * post_perovskite (ppv) * bridgmanite (pv) * ringwoodite (ri) * mg_fe_aluminous_spinel (sp) * wadsleyite (wa)

Relaxed solutions (which account for softening through the spin transition):

- ferropericlase_relaxed
- bridgmanite_relaxed

Endmember phases: * albite (ab) * acmite (acm) * almandine (al) * al_perovskite (alpv) * anorthite (an) * alpha_nao2_phase (anao) * grossular (andr) * alpha_pbo_2_sio_2 (apbo) * al_post_perovskite (appv) * ca_perovskite (capv) * lime_tschermak (cats) * clinoenstatite (cen) * corundum (co) * coesite (coes) * cr_post_perovskite (cppv) * cr_ca_ferrite (crcf) * cr_perovskite (crpv) * diopside (di) * enstatite (en) * eskolaite (esk) * fayalite (fa) * fealo3_perovskite_hs (fapv) * alpha_(bcc)_iron (fea) * hp_clinoferrosilite (fec2) * fe_ca_ferrite (fecf) * epsilon_(hcp)_iron (fee) * gamma_(fcc)_iron (feg) * fe_akimotoite (feil) * fe_perovskite (fepv) * fe_ringwoodite (feri) * fe_wadsleyite (fewa) * fe_nal_phase (fnal) * forsterite (fo) * fe_postperovskite (fppv) * ferrosilite (fs) * grossular (gr) * hercynite (hc) * hedenbergite (he) * hematite (hem) * fe2o3_perovskite_hs (hepv) * fe2o3_perovskite_ls (hlpv) * high_pressure_magnetit (hmag) * hs_fe2o3_post_perovski (hppv) * jadeite (jd) * knorringite (knor) * kyanite (ky) * ls_fe2o3_post_perovski (lppv) * magnetite (mag) * hp_clinoenstatite (mgc2) * mg_ca_ferrite (mgcf) * mg_akimotoite (mgil) * mg_majorite (mgmj) * mg_perovskite (mgpv) * mg_ringwoodite (mgri) * mg_tschermaks (mgts) * mg_wadsleyite (mgwa) * mg_nal_phase (mnal) * mg_postperovskite (mppv) * na_ca_ferrite (nacf) * na_majorite (namj) * nepheline (neph) * na_nal_phase (nnal) * ortho_diopside (odi) * periclase (pe) * pircochromite (picr) * pseudo_wollastonite (pwo) * pyrope (py) * quartz (qtz) * magnetite (smag) * spinel (sp) * stishovite (st) * wollastonite (wo) * wustite (wu) * wustite_low_spin (wuls)

File autogenerated using HeFESTO_2024_to_burnman.py and HeFESTo_parameters_010123.

class burnman.minerals.SLB_2024.**ab**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**acm**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**al**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**alpv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**an**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**anao**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**andr**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**apbo**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**appv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**capv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**cats**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**cen**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**co**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**coes**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**cppv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**crcf**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**crpv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**di**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**en**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**esk**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fa**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fapv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fea**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fec2**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fecf**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fee**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**feg**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**feil**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fepv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**feri**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fewa**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fnal**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fo**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fppv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**fs**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**gr**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hc**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**he**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hem**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hepv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hlpv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hmag**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**hppv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**jd**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**knor**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**ky**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**lppv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mag**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgc2**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgcf**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgil**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgmj**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgpv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgri**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgts**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mgwa**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mnal**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**mppv**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**nacf**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**namj**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**neph**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**nnal**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**odi**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**pe**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**picr**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**pwo**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**py**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**qtz**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**smag**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**sp**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**SLB3Stishovite**

Bases: *SLB3*

shear_modulus(*pressure*, *temperature*, *volume*, *params*)

Returns the shear modulus. $[Pa]$ SLB2024 first calculate the shear modulus at T_0 using the third order Birch-Murnaghan equation of state, then apply the softening relevant to the temperature and pressure of interest, and then apply the usual thermal term.

class burnman.minerals.SLB_2024.**st**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**wo**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**wu**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**wuls**

Bases: *Mineral*

class burnman.minerals.SLB_2024.**c2c_pyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**calcium_ferrite_structured_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**clinopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**garnet**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**ilmenite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**ferropericlase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**ferropericlase_relaxed**

Bases: *RelaxedSolution*

class burnman.minerals.SLB_2024.**new_aluminous_phase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**olivine**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**orthopyroxene**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**plagioclase**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**post_perovskite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**bridgmanite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**bridgmanite_relaxed**

Bases: *RelaxedSolution*

class burnman.minerals.SLB_2024.**ringwoodite**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**mg_fe_aluminous_spinel**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.SLB_2024.**wadsleyite**(*molar_fractions=None*)

Bases: *Solution*

burnman.minerals.SLB_2024.**albite**

alias of ab

burnman.minerals.SLB_2024.**acmite**

alias of acm

burnman.minerals.SLB_2024.**almandine**

alias of al

burnman.minerals.SLB_2024.**al_perovskite**

alias of alpv

burnman.minerals.SLB_2024.**anorthite**

alias of an

burnman.minerals.SLB_2024.**alpha_nao2_phase**

alias of anao

burnman.minerals.SLB_2024.**alpha_pbo_2_sio_2**

alias of apbo

burnman.minerals.SLB_2024.**al_post_perovskite**

alias of appv

burnman.minerals.SLB_2024.**ca_perovskite**

alias of capv

burnman.minerals.SLB_2024.**lime_tschermak**
alias of cats

burnman.minerals.SLB_2024.**clinoenstatite**
alias of cen

burnman.minerals.SLB_2024.**corundum**
alias of co

burnman.minerals.SLB_2024.**coesite**
alias of coes

burnman.minerals.SLB_2024.**cr_post_perovskite**
alias of cppv

burnman.minerals.SLB_2024.**cr_ca_ferrite**
alias of crcf

burnman.minerals.SLB_2024.**cr_perovskite**
alias of crpv

burnman.minerals.SLB_2024.**diopside**
alias of di

burnman.minerals.SLB_2024.**enstatite**
alias of en

burnman.minerals.SLB_2024.**eskolaite**
alias of esk

burnman.minerals.SLB_2024.**fayalite**
alias of fa

burnman.minerals.SLB_2024.**fealo3_perovskite_hs**
alias of fapv

burnman.minerals.SLB_2024.**alpha_bcc_iron**
alias of fea

burnman.minerals.SLB_2024.**hp_clinoferrosilite**
alias of fec2

burnman.minerals.SLB_2024.**fe_ca_ferrite**
alias of fecf

burnman.minerals.SLB_2024.**epsilon_hcp_iron**
alias of fee

burnman.minerals.SLB_2024.**gamma_fcc_iron**
alias of feg

burnman.minerals.SLB_2024.**fe_akimotoite**
alias of feil

burnman.minerals.SLB_2024.**fe_perovskite**
alias of fepv

burnman.minerals.SLB_2024.**fe_ringwoodite**
alias of feri

burnman.minerals.SLB_2024.**fe_wadsleyite**
alias of fewa

burnman.minerals.SLB_2024.**fe_nal_phase**
alias of fnal

burnman.minerals.SLB_2024.**forsterite**
alias of fo

burnman.minerals.SLB_2024.**fe_postperovskite**
alias of fppv

burnman.minerals.SLB_2024.**ferrosilite**
alias of fs

burnman.minerals.SLB_2024.**grossular**
alias of gr

burnman.minerals.SLB_2024.**hercynite**
alias of hc

burnman.minerals.SLB_2024.**hedenbergite**
alias of he

burnman.minerals.SLB_2024.**hematite**
alias of hem

burnman.minerals.SLB_2024.**fe2o3_perovskite_hs**
alias of hepv

burnman.minerals.SLB_2024.**fe2o3_perovskite_ls**
alias of hlpv

burnman.minerals.SLB_2024.**high_pressure_magnetit**
alias of hmag

burnman.minerals.SLB_2024.**hs_fe2o3_post_perovski**
alias of hppv

burnman.minerals.SLB_2024.**jadeite**
alias of jd

burnman.minerals.SLB_2024.**knorringite**
alias of knor

burnman.minerals.SLB_2024.**kyanite**
alias of ky

burnman.minerals.SLB_2024.**ls_fe2o3_post_perovski**
alias of lppv

burnman.minerals.SLB_2024.**hp_clinoenstatite**
alias of mgc2

burnman.minerals.SLB_2024.**mg_ca_ferrite**
alias of mgcf

burnman.minerals.SLB_2024.**mg_akimotoite**
alias of mgil

burnman.minerals.SLB_2024.**mg_majorite**
alias of mgmj

burnman.minerals.SLB_2024.**mg_perovskite**
alias of mgpv

burnman.minerals.SLB_2024.**mg_ringwoodite**
alias of mgri

burnman.minerals.SLB_2024.**mg_tschermaks**
alias of mgts

burnman.minerals.SLB_2024.**mg_wadsleyite**
alias of mgwa

burnman.minerals.SLB_2024.**mg_nal_phase**
alias of mnal

burnman.minerals.SLB_2024.**mg_postperovskite**
alias of mppv

burnman.minerals.SLB_2024.**na_ca_ferrite**
alias of nacf

burnman.minerals.SLB_2024.**na_majorite**
alias of namj

burnman.minerals.SLB_2024.**nepheline**
alias of neph

burnman.minerals.SLB_2024.**na_nal_phase**
alias of nnal

burnman.minerals.SLB_2024.**ortho_diopside**
alias of odi

burnman.minerals.SLB_2024.**periclase**
alias of pe

burnman.minerals.SLB_2024.**pircochromite**
alias of picr

burnman.minerals.SLB_2024.**pseudo_wollastonite**
alias of pwo

burnman.minerals.SLB_2024.**pyrope**
alias of py

burnman.minerals.SLB_2024.**quartz**
alias of qtz

burnman.minerals.SLB_2024.**magnetite**
alias of smag

burnman.minerals.SLB_2024.**spinel**
alias of sp

burnman.minerals.SLB_2024.**stishovite**
alias of st

burnman.minerals.SLB_2024.**wollastonite**
alias of wo

burnman.minerals.SLB_2024.**wustite**
alias of wu

burnman.minerals.SLB_2024.**wustite_low_spin**
SOLUTION ALIASES

burnman.minerals.SLB_2024.**c2c**
alias of c2c_pyroxene

burnman.minerals.SLB_2024.**cf**
alias of calcium_ferrite_structured_phase

burnman.minerals.SLB_2024.**cpx**
alias of clinopyroxene

burnman.minerals.SLB_2024.**gt**
alias of garnet

burnman.minerals.SLB_2024.**il**
alias of ilmenite

burnman.minerals.SLB_2024.**mw**
alias of ferropericlase

burnman.minerals.SLB_2024.**nal**
alias of new_aluminous_phase

burnman.minerals.SLB_2024.**ol**
alias of olivine

burnman.minerals.SLB_2024.**opx**
alias of orthopyroxene

burnman.minerals.SLB_2024.**plg**
alias of plagioclase

burnman.minerals.SLB_2024.**ppv**
alias of post_perovskite

burnman.minerals.SLB_2024.**pv**
alias of bridgmanite

burnman.minerals.SLB_2024.**ri**
alias of ringwoodite

burnman.minerals.SLB_2024.**wa**
alias of wadsleyite

Sundman 1991 models for iron Written in the Holland and Powell, 2011 form

EoS terms for bcc are from HP_2011_ds62 for iron EoS terms for fcc are from an unpublished calibration (R. Myhill, 02/2017)

class burnman.minerals.Sundman_1991.**bcc_iron**
Bases: *Mineral*

class burnman.minerals.Sundman_1991.**fcc_iron**
Bases: *Mineral*

20.20.18 ig50NCKFMASHTOCr

HPx-eos solutions using endmembers from dataset HGP_2018_ds633.

Contains the following solutions: * pl4tr * k4tr * pli * plc * ol * ksp * mu * bi * g * ep * cd * opx * cpx * spn * hb * ilm

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.ig50NCKFMASHTOCr.**pl4tr**(*molar_fractions=None*)
Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**k4tr**(*molar_fractions=None*)
Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**pli**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**plc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**ol**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**ksp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**mu**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**bi**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**ep**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**cd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**cpx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**spn**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**hb**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASHTOCr.**ilm**(*molar_fractions=None*)

Bases: *Solution*

20.20.19 ig50NCKFMASTOCr

HPx-eos solutions using endmembers from dataset HGP_2018_ds633.

Contains the following solutions: * pl4tr * k4tr * pli * plc * ol * ksp * g * opx * cpx * spn * ilm

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.ig50NCKFMASTOCr.**pl4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**k4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**pli**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**plc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**ol**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**ksp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**cpx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**spn**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.ig50NCKFMASTOCr.**ilm**(*molar_fractions=None*)

Bases: *Solution*

20.20.20 mb50NCKFMASHTO

HPx-eos solutions using endmembers from dataset HP_2011_ds62.

Contains the following solutions: * hb * aug * dio * opx * g * ol * pl4tr * abc * k4tr * ksp * plc * pli * sp * ilm * ilmm * ep * bi * mu * chl

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.mb50NCKFMASHTO.**hb**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**aug**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**dio**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**ol**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**pl4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**abc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**k4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**ksp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**plc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**pli**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**sp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**ilm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**ilmm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**ep**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**bi**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**mu**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mb50NCKFMASHTO.**chl**(*molar_fractions=None*)

Bases: *Solution*

20.20.21 mp50KFMASH

HPx-eos solutions using endmembers from dataset HP_2011_ds62.

Contains the following solutions: * g * mu * bi * opx * sa * cd * st * chl * ctd * sp1

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.mp50KFMASH.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**mu**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**bi**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**sa**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**cd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**st**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**chl**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**ctd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50KFMASH.**sp1**(*molar_fractions=None*)

Bases: *Solution*

20.20.22 mp50MnNCKFMASHTO

HPx-eos solutions using endmembers from dataset HP_2011_ds62.

Contains the following solutions: * g * pl4tr * k4tr * plc * ksp * ep * ma * mu * bi * opx * sa * cd * st * chl * ctd * sp * ilmm * ilm * mt1

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.mp50MnNCKFMASHTO.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**pl4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**k4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**plc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ksp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ep**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ma**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**mu**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**bi**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**sa**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**cd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**st**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**chl**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ctd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**sp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ilmm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**ilm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50MnNCKFMASHTO.**mt1**(*molar_fractions=None*)

Bases: *Solution*

20.20.23 mp50NCKFMASHTO

HPx-eos solutions using endmembers from dataset HP_2011_ds62.

Contains the following solutions: * g * pl4tr * k4tr * plc * ksp * ep * ma * mu * bi * opx * sa * cd * st * chl * ctd * sp * ilmm * ilm * mt1

The values in this document are all in S.I. units, unlike those in the original THERMOCALC file.

This file is autogenerated using hpx_eos_to_burnman.py

class burnman.minerals.mp50NCKFMASHTO.**g**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**pl4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**k4tr**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**plc**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ksp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ep**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ma**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**mu**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**bi**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**opx**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**sa**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**cd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**st**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**chl**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ctd**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**sp**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ilmm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**ilm**(*molar_fractions=None*)

Bases: *Solution*

class burnman.minerals.mp50NCKFMASHTO.**mt1**(*molar_fractions=None*)

Bases: *Solution*

20.20.24 Other minerals

class burnman.minerals.other.**liquid_iron**

Bases: *Mineral*

Liquid iron equation of state from Anderson and Ahrens (1994)

class burnman.minerals.other.**ZSB_2013_mg_perovskite**

Bases: *Mineral*

class burnman.minerals.other.**ZSB_2013_fe_perovskite**

Bases: *Mineral*

class burnman.minerals.other.**Speziale_fe_periclase**

Bases: *HelperSpinTransition*

class burnman.minerals.other.**Speziale_fe_periclase_HS**

Bases: *Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Speziale_fe_periclase_LS**

Bases: *Mineral*

Speziale et al. 2007, Mg#=83

class burnman.minerals.other.**Liquid_Fe_Anderson**

Bases: *Mineral*

Anderson & Ahrens, 1994 JGR

class burnman.minerals.other.**Fe_Dewaele**

Bases: *Mineral*

Dewaele et al., 2006, Physical Review Letters

20.21 Calibrant databases

This package contains a large collection of calibrant models. These calibrants are all derived from `burnman.Calibrant`, and share the same interface. The models are taken from a variety of published sources:

- *Anderson_1989*
- *Armentrout_2015*
- *Campbell_2009*
- *Chidester_2021*
- *Decker_1971*
- *Dewaele_2008*
- *Dewaele_2012*
- *Dewaele_2013*
- *Dewaele_2020*
- *Dorogokupets_2007*
- *Dorogokupets_2017*
- *Dubrovinsky_1998*
- *Fei_2007*
- *Fei_2016*
- *Holmes_1989*
- *Huang_2016*
- *LeGodec_2000*
- *Litasov_2013*
- *Matsui_2009*
- *Matsui_2010*
- *Matsui_2012*
- *Miozzi_2020*
- *Ono_2022*
- *Pigott_2015*
- *Shi_2022*
- *Shim_2002*
- *Sokolova_2013*

- *Speziale_2001*
- *Tange_2009*
- *Tateno_2019*
- *Tsuchiya_2003*
- *Walker_2002*
- *Zeng_2010*
- *Zha_2004*
- *Zha_2008*
- *Zhang_1999*
- *Zhao_1997*
- *Zhao_2000*

20.21.1 Anderson_1989

class burnman.calibrants.Anderson_1989.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Anderson et al. (1989; https://doi.org/10.1063/1.342969).

20.21.2 Armentrout_2015

class burnman.calibrants.Armentrout_2015.**Co_fcc**

Bases: *Calibrant*

The FCC Cobalt pressure standard reported by Armentrout (2015; https://doi.org/10.1063/1.4935087).

20.21.3 Campbell_2009

class burnman.calibrants.Campbell_2009.**Ni_fcc**

Bases: *Calibrant*

The FCC Ni pressure standard reported by Campbell (2009; https://doi.org/10.1016/j.epsl.2009.07.022).

20.21.4 Chidester_2021

class burnman.calibrants.Chidester_2021.**KCl_B2**

Bases: *Calibrant*

The B2 KCl pressure standard reported by Chidester et al. (2021; https://doi.org/10.1103/PhysRevB.104.094107).

20.21.5 Decker_1971

class burnman.calibrants.Decker_1971.**NaCl_B1**

Bases: *Calibrant*

The NaCl (B1 structured) pressure standard reported by Decker (1971; https://doi.org/10.1063/1.1660714).

> **Note**
>
> This calibrant is not exactly the same as that proposed by Decker. The cold compression curve has here been approximated by a 4th order Birch-Murnaghan EoS, as described in Matsui et al. (2012; https://doi.org/10.2138/am.2012.4136).

20.21.6 Dewaele_2008

class burnman.calibrants.Dewaele_2008.**Ni**

Bases: *Calibrant*

The Ni pressure standard reported by Dewaele (2008; Table IV; https://doi.org/10.1103/PhysRevB.78.104102).

20.21.7 Dewaele_2012

class burnman.calibrants.Dewaele_2012.**KBr_B2**

Bases: *Calibrant*

The B2 KBr pressure standard reported by Dewaele et al. (2012; https://doi.org/10.1103/PhysRevB.85.214105).

class burnman.calibrants.Dewaele_2012.**KCl_B2**

Bases: *Calibrant*

The B2 KCl pressure standard reported by Dewaele et al. (2012; https://doi.org/10.1103/PhysRevB.85.214105).

20.21.8 Dewaele_2013

class burnman.calibrants.Dewaele_2013.**Al2O3_corundum**

Bases: *Calibrant*

The alpha-Al2O3 (corundum) pressure standard reported by Dewaele and Torrent (2013; https://doi.org/10.1103/PhysRevB.88.064107). The thermal model is from Shi (2022).

20.21.9 Dewaele_2020

class burnman.calibrants.Dewaele_2020.**CsCl**

Bases: *Calibrant*

The CsCl pressure standard reported by Dewaele (2020; https://doi.org/10.1080/08957959.2020.1774754). The thermal model is from Decker (1971).

20.21.10 Dorogokupets_2007

class burnman.calibrants.Dorogokupets_2007.**Ag**

Bases: *Calibrant*

The Ag pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**Al**

Bases: *Calibrant*

The Al pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**Cu**

Bases: *Calibrant*

The Cu pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**diamond**

Bases: *Calibrant*

The diamond pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**MgO**

Bases: *Calibrant*

The MgO pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**Pt**

Bases: *Calibrant*

The Pt pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**Ta**

Bases: *Calibrant*

The Ta pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

class burnman.calibrants.Dorogokupets_2007.**W**

Bases: *Calibrant*

The W pressure standard reported by Dorogokupets and Oganov (2007; https://doi.org/10.1103/PhysRevB.75.024115).

20.21.11 Dorogokupets_2017

class burnman.calibrants.Dorogokupets_2017.**Fe_bcc**

Bases: *Calibrant*

The BCC Fe pressure standard reported by Dorogokupets (2017; https://doi.org/10.1038/srep41863).

class burnman.calibrants.Dorogokupets_2017.**Fe_fcc**

Bases: *Calibrant*

The FCC Fe pressure standard reported by Dorogokupets (2017; https://doi.org/10.1038/srep41863).

class burnman.calibrants.Dorogokupets_2017.**Fe_hcp**

Bases: *Calibrant*

The HCP Fe pressure standard reported by Dorogokupets (2017; https://doi.org/10.1038/srep41863).

20.21.12 Dubrovinsky_1998

class burnman.calibrants.Dubrovinsky_1998.**Al2O3_corundum**

Bases: *Calibrant*

The Al2O3 pressure standard reported by Dubrovinsky (1998; https://dx.doi.org/10.1007/s002690050133).

20.21.13 Fei_2007

class burnman.calibrants.Fei_2007.**Pt**

Bases: *Calibrant*

The Pt pressure standard reported by Fei et al. (2007; https://doi.org/10.1073/pnas.0609013104).

class burnman.calibrants.Fei_2007.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Fei et al. (2007; https://doi.org/10.1073/pnas.0609013104).

20.21.14 Fei_2016

class burnman.calibrants.Fei_2016.**Fe_hcp**

Bases: *Calibrant*

The HCP Fe pressure standard reported by Fei et al. (2016; https://doi.org/10.1002/2016GL069456).

20.21.15 Holmes_1989

class burnman.calibrants.Holmes_1989.**Pt**

Bases: *Calibrant*

The Pt pressure standard reported by Holmes et al. (1989; https://doi.org/10.1063/1.344177).

20.21.16 Huang_2016

class burnman.calibrants.Huang_2016.**Mo**

Bases: *Calibrant*

The Mo pressure standard reported by Huang et al. (2016; https://doi.org/10.1038/srep19923).

20.21.17 LeGodec_2000

class burnman.calibrants.LeGodec_2000.**hBN**

Bases: *Calibrant*

The hexagonal boron nitride pressure standard reported by LeGodec et al. (2000; http://dx.doi.org/10.1080/08957950008200304).

20.21.18 Litasov_2013

class burnman.calibrants.Litasov_2013.**Mo_bcc**

Bases: *Calibrant*

The BCC Mo pressure standard reported by Litasov et al. (2013; https://doi.org/10.1063/1.4794127).

class burnman.calibrants.Litasov_2013.**W_bcc**

Bases: *Calibrant*

The BCC W pressure standard reported by Litasov et al. (2013; https://doi.org/10.1063/1.4799018).

20.21.19 Matsui_2009

class burnman.calibrants.Matsui_2009.**Pt**

Bases: *Calibrant*

The Pt pressure standard reported by Matsui (2009; https://doi.org/10.1063/1.3054331).

20.21.20 Matsui_2010

class burnman.calibrants.Matsui_2010.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Matsui (2010; https://dx.doi.org/10.1088/1742-6596/215/1/012197).

20.21.21 Matsui_2012

class burnman.calibrants.Matsui_2012.**NaCl_B1**

Bases: *Calibrant*

The NaCl (B1 structured) pressure standard reported by Matsui (2012; https://doi.org/10.2138/am.2012.4136).

20.21.22 Miozzi_2020

class burnman.calibrants.Miozzi_2020.**Fe_hcp**

Bases: *Calibrant*

The HCP Fe pressure standard reported by Miozzi (2020; https://doi.org/10.3390/min10020100).

20.21.23 Ono_2022

class burnman.calibrants.Ono_2022.**Re**

Bases: *Calibrant*

The Re pressure standard reported by Ono (2022; https://doi.org/10.1155/2022/7545777).

20.21.24 Pigott_2015

class burnman.calibrants.Pigott_2015.**Ni**

Bases: *Calibrant*

The Ni pressure standard reported by Pigott (2015; https://doi.org/10.1002/2015GL066577).

20.21.25 Shi_2022

class burnman.calibrants.Shi_2022.**Al2O3_corundum**

Bases: *Calibrant*

The corundum-structured Al2O3 pressure standard reported by Shi (2022; https://doi.org/10.1029/2021JB023805).

20.21.26 Shim_2002

class burnman.calibrants.Shim_2002.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Shim et al. (2002; https://doi.org/10.1016/S0012-821X(02)00917-2).

20.21.27 Sokolova_2013

class burnman.calibrants.Sokolova_2013.**Mo**

Bases: *Calibrant*

The Mo pressure standard reported by Sokolova et al. (2013), as reported by Huang et al. (2016; https://doi.org/10.1038/srep19923).

20.21.28 Speziale_2001

class burnman.calibrants.Speziale_2001.**MgO**

Bases: *Calibrant*

The MgO pressure standard reported by Speziale (2001; https://doi.org/10.1029/2000JB900318).

20.21.29 Tange_2009

class burnman.calibrants.Tange_2009.**MgO_BM3**

Bases: *Calibrant*

The MgO pressure standard reported by Tange et al. (2009; https://doi.org/10.1029/2008JB005813). BM3 version.

class burnman.calibrants.Tange_2009.**MgO_Vinet**

Bases: *Calibrant*

The MgO pressure standard reported by Tange et al. (2009; https://doi.org/10.1029/2008JB005813). Vinet version.

20.21.30 Tateno_2019

class burnman.calibrants.Tateno_2019.**KCl_B2**

Bases: *Calibrant*

The B2 KCl pressure standard reported by Tateno (2019; https://doi.org/10.2138/am-2019-6779).

20.21.31 Tsuchiya_2003

class burnman.calibrants.Tsuchiya_2003.**Au**

Bases: *Calibrant*

The Au pressure standard reported by Tsuchiya (2003; https://doi.org/10.1029/2003JB002446).

20.21.32 Walker_2002

class burnman.calibrants.Walker_2002.**KCl_B2**

Bases: *Calibrant*

The B2 KCl pressure standard reported by Walker (2002; https://doi.org/10.2138/am-2002-0701).

20.21.33 Zeng_2010

class burnman.calibrants.Zeng_2010.**Mo**

Bases: *Calibrant*

The Mo pressure standard from Zeng (2010), as reported by Huang et al. (2016; https://doi.org/10.1038/srep19923).

20.21.34 Zha_2004

class burnman.calibrants.Zha_2004.**Re**

Bases: *Calibrant*

The Re pressure standard reported by Zha et al. (2004; https://doi.org/10.1063/1.1765752).

20.21.35 Zha_2008

class burnman.calibrants.Zha_2008.**Pt**

Bases: *Calibrant*

The Pt pressure standard reported by Zha (2008; https://doi.org/10.1063/1.2844358).

20.21.36 Zhang_1999

class burnman.calibrants.Zhang_1999.**Fe_bcc**

Bases: *Calibrant*

The BCC Fe pressure standard reported by Zhang et al. (1999; https://dx.doi.org/10.1007/s002690050178).

20.21.37 Zhao_1997

class burnman.calibrants.Zhao_1997.**hBN**

Bases: *Calibrant*

The hBN pressure standard reported by Zhao et al. (1997; https://doi.org/10.1080/08957959708240481).

20.21.38 Zhao_2000

class burnman.calibrants.Zhao_2000.**Mo**

Bases: *Calibrant*

The Mo pressure standard reported by Zhao et al. (2000; https://doi.org/10.1103/PhysRevB.62.8766).

burnman.calibrants.tools.**pressure_to_pressure**(*old_calibrant*, *new_calibrant*, *pressure*, *temperature*, *PT_covariance=None*)

Convert from pressure defined by one calibrated equation of state of a material to pressure defined by an independent calibration of the same material.

Parameters

- **old_calibrant** (*burnman.Calibrant*) – The original calibration used to estimate the pressure
- **new_calibrant** (*burnman.Calibrant*) – The new calibration from which the pressure is desired
- **pressure** (*float*) – The pressure calculated using the old calibration
- **temperature** (*float*) – The temperature of the material
- **PT_covariance** (*2x2 numpy.array [optional]*) – The pressure-temperature variance-covariance matrix

Returns

The pressure of the calibrant [Pa] (float) and a 2x2 numpy array (if the PT_covariance is provided) containing the pressure-temperature variance-covariance matrix.

Return type
tuple

20.22 Optimization

class burnman.optimize.composition_fitting.**DummyCompositionSolution**(*endmember_element_formulae*, *endmember_site_formulae*)

Bases: *Solution*

A mock Solution class to enable compositional fitting when no thermodynamic model is available. Useful for pure composition analysis.

The model is initialized with appropriate chemical formulae for each endmember, and can do all basic compositional processing that doesn't involve any material properties.

Parameters

- **endmember_element_formulae** (*list of str*) – Formulae for each of the independent endmembers. e.g. ['Mg2SiO4', 'Fe2SiO4'].
- **endmember_site_formulae** (*list of str*) – Site formulae for each of the independent endmembers, in the same order as endmember_element_formulae. e.g. ['[Mg]2SiO4', '[Fe]2SiO4'].

property endmember_formulae

A list of formulae for all the endmembers in the solution.

property C_p

Alias for *heat_capacity_p()*

property C_v

Alias for *heat_capacity_v()*

property G

Alias for *shear_modulus()*

property G_eff

Alias for *effective_shear_modulus()*

property H

Alias for *enthalpy()*

property K_S

Alias for *isentropic_bulk_modulus_reuss()*

property K_T

Alias for *isothermal_bulk_modulus_reuss()*

property K_eff

Alias for *effective_isentropic_bulk_modulus()*

property P

Alias for *pressure()*

property S

Alias for *entropy()*

property T

Alias for *temperature()*

property V

Alias for *volume()*

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 alpha

Alias for *thermal_expansivity()*

property beta_S

Alias for *isentropic_compressibility_reuss()*

property beta_T

Alias for *isothermal_compressibility_reuss()*

property bulk_sound_velocity

Returns bulk sound speed of the solution [m/s]. Aliased with self.v_phi.

property compositional_basis

Returns

A basis for the compositional degrees of freedom, orthogonal to the reaction space.

Each row of the returned array defines an independent compositional constraint in terms of moles of endmembers. These constraints span the row space of the stoichiometric matrix and are complementary to the reaction basis (which spans the left null space).

For example, endmembers with site-species occupancies [Fe][Mg] and [Mg][Fe] have reaction basis vector [1, -1]. A compositional basis vector orthogonal to this is: [1, 1], which satisfies the requirement that the Fe-Mg ratio of the system remains fixed and the total amount of the two sites remains equal.

Return type

numpy.ndarray

property compositional_null_basis

Returns a basis for the compositional nullspace (kernel) of the stoichiometric matrix.

This basis consists of vectors that represent linear constraints any valid bulk composition must satisfy to be formed from the endmembers. In other words, for any bulk composition vector b

that can be expressed as a combination of the endmembers, multiplying it by these vectors results in zero ($N @ b = 0$).

These constraints typically reflect conserved properties such as elemental mass balance or charge neutrality.

Example: Suppose the stoichiometric matrix corresponds to elements C and O in species, and the nullspace basis vector is *[1, -2]*. Then for any bulk composition vector $b = [b_C, b_O]$, the constraint *1 * b_C - 2 * b_O = 0* must hold. This means the ratio of carbon to oxygen must satisfy that relationship to be achievable from the given endmembers.

Returns
A 2D array where each row is a basis vector of the nullspace.

Return type
numpy.ndarray

`copy()`

`debug_print(indent='')`

Print a human-readable representation of this Material.

`property density`

Returns density of the solution [kg/m^3]. Aliased with self.rho.

`property dependent_element_indices`

The element indices not included in the independent list.

`property effective_isentropic_bulk_modulus`

Returns the effective isentropic bulk modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *K_eff()*.

Returns
Effective isentropic bulk modulus in [Pa].

Return type
float

`property effective_shear_modulus`

Returns the effective shear modulus of the mineral.

> **Note**
> Needs to be implemented in derived classes. Aliased with *G_eff()*.

Returns
Effective shear modulus in [Pa].

Return type
float

property elements

A list of the elements which could be contained in the solution, returned in the IUPAC element order.

property endmember_names

A list of names for all the endmembers in the solution.

property endmembers

property enthalpy

Extensive enthalpy of the material (J).

property entropy

Extensive entropy of the material (J/K).

property entropy_hessian

Returns an array containing the second compositional derivative of the entropy [J/K]. Property specific to solutions.

evaluate(*vars_list*, *pressures*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given pressure and temperature conditions. At the end it resets the set_state to the original values. The user needs to call set_method() before.

Parameters

- **vars_list** (*list of strings*) – Variables to be returned for given conditions
- **pressures** (numpy.array, n-dimensional) – ndlist or ndarray of float of pressures in [Pa].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

evaluate_with_volumes(*vars_list*, *volumes*, *temperatures*, *molar_fractions=None*)

Returns an array of material properties requested through a list of strings at given volume 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
- **volumes** (numpy.array, n-dimensional) – ndlist or ndarray of float of volumes in [m^3].
- **temperatures** (numpy.array, n-dimensional) – ndlist or ndarray of float of temperatures in [K].

Returns
List or array returning all variables at given pressure/temperature values. output[i][j] is property vars_list[j] for temperatures[i] and pressures[i]. Attempts to return an array, falls back to a list if the returned properties have different shapes.

Return type
list or numpy.array, n-dimensional

property excess_enthalpy
Returns excess molar enthalpy [J/mol]. Property specific to solutions.

property excess_entropy
Returns excess molar entropy [J/K/mol]. Property specific to solutions.

property excess_gibbs
Returns molar excess gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_entropies
Returns excess partial entropies [J/K]. Property specific to solutions.

property excess_partial_gibbs
Returns excess partial molar gibbs free energy [J/mol]. Property specific to solutions.

property excess_partial_volumes
Returns excess partial volumes [m^3]. Property specific to solutions.

property excess_volume
Returns excess molar volume of the solution [m^3/mol]. Specific property for solutions.

property formula
Returns molar chemical formula of the solution. :rtype: Counter

property gibbs
Extensive Gibbs free energy of the material (J).

property gibbs_hessian
Returns an array containing the second compositional derivative of the Gibbs free energy [J]. Property specific to solutions.

property gr
Alias for *grueneisen_parameter()*

property grueneisen_parameter
Returns grueneisen parameter of the solution [unitless]. Aliased with self.gr.

property heat_capacity_p

Extensive heat capacity at constant pressure of the material (J/K).

property heat_capacity_v

Extensive heat capacity at constant volume of the material (J/K).

property helmholtz

Extensive Helmholtz free energy of the material (J).

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 internal_energy

Extensive internal energy of the material (J).

property isentropic_bulk_modulus_reuss

Returns adiabatic bulk modulus of the solution [Pa]. Aliased with self.K_S.

property isentropic_compressibility_reuss

Returns adiabatic compressibility of the solution. (or inverse adiabatic bulk modulus) [1/Pa]. Aliased with self.K_S.

property isentropic_thermal_gradient

(copied from isentropic_thermal_gradient):

Returns
dTdP, the change in temperature with pressure at constant entropy [Pa/K]

Return type
float

property isothermal_bulk_modulus_reuss

Returns isothermal bulk modulus of the solution [Pa]. Aliased with self.K_T.

property isothermal_compressibility_reuss

Returns isothermal compressibility of the solution. (or inverse isothermal bulk modulus) [1/Pa]. Aliased with self.K_T.

property mass

Returns the mass of this material.

Returns
Mass in [kg].

Return type
float

property molar_enthalpy

Returns molar enthalpy of the solution [J/mol]. Aliased with self.H.

property molar_entropy

Returns molar entropy of the solution [J/K/mol]. Aliased with self.S.

property molar_gibbs

Returns molar Gibbs free energy of the solution [J/mol]. Aliased with self.gibbs.

property molar_heat_capacity_p

Returns molar heat capacity at constant pressure of the solution [J/K/mol]. Aliased with self.C_p.

property molar_heat_capacity_v

Returns molar heat capacity at constant volume of the solution [J/K/mol]. Aliased with self.C_v.

property molar_helmholtz

Returns molar Helmholtz free energy of the solution [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 solution [kg/mol].

property molar_volume

Returns molar volume of the solution [m^3/mol]. Aliased with self.V.

property n_endmembers

The number of endmembers in the solution.

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 number_of_moles

Returns the number of moles in this material.

Returns
Number of moles.

Return type
float

property p_wave_velocity

Returns P wave speed of the solution [m/s]. Aliased with self.v_p.

property partial_entropies

Returns endmember partial entropies [J/K]. Property specific to solutions.

property partial_gibbs

Returns endmember partial molar gibbs free energy [J/mol]. Property specific to solutions.

property partial_volumes

Returns endmember partial volumes [m^3]. Property specific to solutions.

property pressure

Returns current pressure that was set with *set_state()*.

> **Note**
>
> Aliased with *P()*.

Returns
Pressure in [Pa].

Return type
float

print_minerals_of_current_state()

Print a human-readable representation of this Material at the current P, T as a list of minerals. This requires set_state() has been called before.

property reaction_basis

Returns a basis for the reaction space, represented as the left nullspace of the stoichiometric matrix.

Each row of the returned array corresponds to a linearly independent chemical reaction that conserves mass and charge. The value at arr[i, j] gives the number of moles of endmember[j] involved in reaction[i].

Returns
An array where each row defines a balanced chemical reaction in terms of molar amounts of endmembers.

Return type
numpy.ndarray

reset()

Resets all cached material properties.

It is typically not required for the user to call this function.

property rho

Alias for *density()*

set_composition(*molar_fractions*)

Set the composition for this solution. Resets cached properties.

Parameters
molar_fractions (*list of float*) – Molar abundance for each endmember, needs to sum to one.

set_method(*method*)

Set the equation of state to be used for this mineral. Takes a string corresponding to any of the predefined equations of state: 'bm3shear2', '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].

set_state_with_volume(*volume*, *temperature*, *pressure_guesses=[0.0, 10000000000.0]*)

This function acts similarly to set_state, but takes volume and temperature as input to find the pressure.

If the mineral is an endmember that has a pressure(T, V) function and does not have property modifiers, this function evaluates the pressure directly. Otherwise, it finds the pressure using the brentq root-finding method on the molar_volume attribute of the mineral. This ensures self-consistency even if the mineral has a pressure-dependent volume modifier.

Parameters

- **volume** (*float*) – The desired molar volume of the mineral [m^3].
- **temperature** (*float*) – The desired temperature of the mineral [K].
- **pressure_guesses** (*list*) – A list of floats denoting the initial low and high guesses for bracketing of the pressure [Pa]. These guesses should preferably bound the correct pressure, but do not need to do so. More importantly, they should not lie outside the valid region of the equation of state. Defaults to [0.e9, 10.e9].

property shear_modulus

Returns shear modulus of the solution [Pa]. Aliased with self.G.

property shear_wave_velocity

Returns shear wave speed of the solution [m/s]. Aliased with self.v_s.

site_formula(*precision=2*, *print_absent_species=True*, *site_occupancies=None*)

Returns the molar chemical formula of the solution with
site occupancies. For example, [Mg0.4Fe0.6]2SiO4.

Parameters

- **precision** (*int*) – Precision with which to print the site occupancies

- **print_absent_species** (*bool*) – If True, prints site occupancies that are absent (i.e., zero).
- **site_occupancies** (*numpy.ndarray*) – Optional site occupancies to use instead of the current ones.

Returns
Molar chemical formula of the solution with site occupancies

Return type
str

property site_occupancies

Returns
The fractional occupancies of species on each site as a flat vector.

Return type
numpy.ndarray

property stoichiometric_array

A numpy array where each element arr[i,j] corresponds to the number of atoms of element[j] in endmember[i].

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 temperature

Returns current temperature that was set with *set_state()*.

> **Note**
> Aliased with *T()*.

Returns
Temperature in [K].

Return type
float

property thermal_expansivity

Returns thermal expansion coefficient (alpha) of the solution [1/K]. Aliased with self.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).

> **Note**
>
> Needs to be implemented in derived classes.

Returns
A list of molar fractions which should sum to 1.0, and a list of *burnman.Mineral* objects containing the minerals in the material.

Return type
tuple

property v_p

Alias for *p_wave_velocity()*

property v_phi

Alias for *bulk_sound_velocity()*

property v_s

Alias for *shear_wave_velocity()*

property volume

Extensive volume of the material (m^3).

property volume_hessian

Returns an array containing the second compositional derivative of the volume [m^3]. Property specific to solutions.

burnman.optimize.composition_fitting.**fit_composition_to_solution**(*solution*, *fitted_variables*, *variable_values*, *variable_covariances*, *variable_conversions=None*, *normalize=True*)

Fit a set of molar fractions to a solution model that best matches observed compositional variables (in a least-squares sense).

The fitting applies appropriate non-negativity constraints (i.e. no species can have a negative occupancy on a site).

Parameters

- **solution** (*burnman.Solution*) – The solution to use in the fitting procedure.
- **fitted_variables** (*list of str*) – A list of the variables used to find the best-fit molar fractions of the solution. These should either be elements such as "Fe", site_species such as "Fef_B" which would correspond to a species labelled Fef on the second site, or user-defined variables which are arithmetic sums of elements and/or site_species defined in "variable_conversions".

- **variable_values** (*numpy.array*) – Numerical values of the fitted variables. These should be given as amounts; they do not need to be normalized.
- **variable_covariances** (*2D numpy.array*) – Covariance matrix of the variables.
- **variable_conversions** (*dict of dict, or None*) – A dictionary converting any user-defined variables into an arithmetic sum of element and site-species amounts. For example, {'Mg_equal': {'Mg_A': 1., 'Mg_B': -1.}}, coupled with Mg_equal = 0 would impose a constraint that the amount of Mg would be equal on the first and second site in the solution.
- **normalize** (*bool, default=True*) – Whether to normalize the molar fractions to sum to one.

Returns

(molar_fractions, covariance_matrix, weighted_residual):

- molar_fractions: Optimized molar fractions of each endmember.
- **covariance_matrix: Covariance matrix of the optimized molar** fractions, computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing** the square root of the weighted sum of squared differences between observed and modeled variables.

Return type

tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

burnman.optimize.composition_fitting.**fit_phase_proportions_to_bulk_composition**(*phase_compositions*, *bulk_composition*)

Determine the proportions of phases that best fit a given bulk composition (in a weighted constrained least squares sense).

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

(phase_proportions, covariance_matrix, weighted_residual)

- phase_proportions: Optimized proportions of each phase.
- **covariance_matrix: Covariance matrix of the optimized phase proportions,** computed from the pseudo-inverse of the weighted design matrix.
- **weighted_residual: The weighted residual of the fit, representing the square root** of the weighted sum of squared differences between observed and modeled bulk compositions.

Return type
tuple of 1D numpy.array, 2D numpy.array and float

See also

burnman.optimize.linear_fitting.weighted_constrained_least_squares() - Performs the weighted constrained least squares optimization used internally.

burnman.optimize.eos_fitting.**default_mle_tolerances**(*material*, *flags*)

Computes default tolerances for maximum likelihood estimation (MLE).

The tolerances are computed per property flag, using either: - A fixed absolute tolerance (e.g., 1.0 J) for energy-related properties - A fixed absolute tolerance (e.g., 100 Pa) for pressure - A relative tolerance (1e-5 x standard state value) for all other properties

All computed tolerances must be > 0.

Parameters

- **material** (*burnman.Material*) – The material instance to evaluate properties on.
- **flags** (*list of str*) – List of property names (strings) for which MLE tolerances should be generated.

Returns
A NumPy array of tolerances, one per property flag.

Return type
np.ndarray

Raises
Exception – If any computed tolerance is not strictly positive.

class burnman.optimize.eos_fitting.**MineralFit**(*mineral*, *data*, *data_covariances*, *flags*, *fit_params*, *mle_tolerances*, *delta_params=None*, *bounds=None*)

Bases: *NonLinearModel*

Class for fitting mineral parameters to experimental data. Instances of this class are passed to burnman.nonlinear_least_squares_fit().

For attributes added to this model when fitting is done, please see the documentation for that function.

`set_params`(*param_values*)

Parameters
param_values (*numpy.ndarray*) – Parameter values to set.

`get_params`()

Returns
Current model parameters as a NumPy array.

Return type
numpy.ndarray

`function`(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Model output.

Return type
numpy.ndarray

`normal`(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Normal vector.

Return type
numpy.ndarray

`validate`()

Ensure that all required model attributes are present before fitting. These include observed data, data covariance matrices, convergence tolerances for MLE projections, and finite difference step sizes.

Raises
AssertionError – If required attributes are missing.

burnman.optimize.eos_fitting.**fit_PTp_data**(*mineral*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a mineral of any type, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the parameters should be optimized.
- **fit_params** (*list of str*) – List of dictionary keys contained in mineral.params corresponding to the variables to be optimized during fitting. Initial guesses are taken from the existing values for the parameters
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional) If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns

Model with optimized parameters.

Return type

burnman.optimize.eos_fitting.MineralFit

burnman.optimize.eos_fitting.**fit_PTV_data**(*mineral*, *fit_params*, *data*, *data_covariances=[]*, *delta_params=None*, *bounds=None*, *param_tolerance=1e-05*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

A simple alias for the fit_PTp_data for when all the data is volume data

class burnman.optimize.eos_fitting.**MineralFitV**(*mineral*, *data*, *data_covariances*, *flags*, *fit_params*, *mle_tolerances*, *delta_params=None*, *bounds=None*)

Bases: *NonLinearModel*

Class for fitting mineral parameters to experimental data using volume as an independent variable. Instances of this class are passed to `burnman.nonlinear_least_squares_fit()`.

For attributes added to this model when fitting is done, please see the documentation for that function.

set_params(*param_values*)

Parameters
param_values (*numpy.ndarray*) – Parameter values to set.

get_params()

Returns
Current model parameters as a NumPy array.

Return type
numpy.ndarray

function(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Model output.

Return type
numpy.ndarray

normal(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Normal vector.

Return type
numpy.ndarray

`validate()`

Ensure that all required model attributes are present before fitting. These include observed data, data covariance matrices, convergence tolerances for MLE projections, and finite difference step sizes.

Raises
`AssertionError` – If required attributes are missing.

`burnman.optimize.eos_fitting.fit_VTp_data(`*mineral*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a mineral of any type, a list of fit parameters and a set of V-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the parameters should be optimized.
- **fit_params** (*list of str*) – List of dictionary keys contained in mineral.params corresponding to the variables to be optimized during fitting. Initial guesses are taken from the existing values for the parameters
- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'P')
- **data** (*2D numpy.array*) – Observed V-T-property values
- **data_covariances** (*3D numpy.array*) – V-T-property covariances (optional) If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the properties.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).

- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns
Model with optimized parameters.

Return type
burnman.optimize.eos_fitting.MineralFitV

burnman.optimize.eos_fitting.**fit_VTP_data**(*mineral*, *fit_params*, *data*, *data_covariances=[]*, *delta_params=None*, *bounds=None*, *param_tolerance=1e-05*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

A simple alias for the fit_VTp_data for when all the data is pressure data

class burnman.optimize.eos_fitting.**SolutionFit**(*solution*, *data*, *data_covariances*, *flags*, *fit_params*, *mle_tolerances*, *delta_params=None*, *bounds=None*)

Bases: *NonLinearModel*

Class for fitting mineral parameters to experimental data. Instances of this class are passed to `burnman.nonlinear_least_squares_fit()`.

For attributes added to this model when fitting is done, please see the documentation for that function.

set_params(*param_values*)

Parameters
param_values (*numpy.ndarray*) – Parameter values to set.

get_params()

Returns
Current model parameters as a NumPy array.

Return type
numpy.ndarray

validate()

Ensure that all required model attributes are present before fitting. These include observed data, data covariance matrices, convergence tolerances for MLE projections, and finite difference step sizes.

Raises
AssertionError – If required attributes are missing.

function(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Model output.

Return type
numpy.ndarray

normal(*x*, *flag*)

Parameters

- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Normal vector.

Return type
numpy.ndarray

burnman.optimize.eos_fitting.**fit_XPTp_data**(*solution*, *fit_params*, *flags*, *data*, *data_covariances=[]*, *mle_tolerances=[]*, *param_tolerance=1e-05*, *delta_params=None*, *bounds=None*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=None*, *verbose=True*)

Given a symmetric solution, a list of fit parameters and a set of P-T-property points and (optional) uncertainties, this function returns a list of optimized parameters and their associated covariances, fitted using the scipy.optimize.curve_fit routine.

Parameters

- **solution** (*burnman.Solution*) – Solution for which the parameters should be optimized.
- **fit_params** (*list of lists*) – Variables to be optimized during fitting. Each list is either of length two or three. The first item of length-2 lists should be a dictionary key contained in one of the endmember mineral.params, and the second item should be the index of the endmember in the solution (indexing starts from 0). The first item of length-3 lists should be one of 'E', 'S' or 'V' (the excess energies, entropies or volumes in each binary). The second two items should be the indices

of the pair of endmembers bounding the binary, in ascending order (indexing starts from 0). Initial guesses are taken from the existing values for the parameters.

- **flags** (*string or list of strings*) – Attribute names for the property to be fit for the whole dataset or each datum individually (e.g. 'V')
- **data** (*2D numpy.array*) – Observed X-P-T-property values
- **data_covariances** (*3D numpy.array*) – X-P-T-property covariances (optional). If not given, all covariance matrices are chosen such that all data points have equal weight, with all error in the pressure.
- **mle_tolerances** (*numpy.array*) – Tolerances for termination of the maximum likelihood iterations (optional).
- **param_tolerance** (*float*) – Fractional tolerance for termination of the nonlinear optimization (optional).
- **delta_params** (*numpy.array*) – Initial values for the change in parameters (optional).
- **bounds** (*2D numpy.array*) – Minimum and maximum bounds for the parameters (optional). The shape must be (n_parameters, 2).
- **max_lm_iterations** (*int*) – Maximum number of Levenberg-Marquardt iterations.
- **verbose** (*bool*) – Whether to print detailed information about the optimization to screen.

Returns

Model with optimized parameters.

Return type

burnman.optimize.eos_fitting.SolutionFit

burnman.optimize.linear_fitting.**weighted_constrained_least_squares**(*A*, *b*, *Cov_b=None*, *equality_constraints=None*, *inequality_constraints=None*, *allow_rank_deficient=False*)

Solves a weighted, constrained least squares problem using cvxpy. The objective function is to minimize the following: sum_squares(Cov_b^(-1/2).A.x - Cov_b^(-1/2).b)) subject to C.x == c D.x <= d

Parameters

- **A** (*2D numpy array*) – An array defining matrix A in the objective function above.
- **b** (*numpy array*) – An array defining vector b in the objective function above.
- **Cov_b** (*2D numpy array*) – A covariance matrix associated with b

- **equality_constraints** (*list containing a 2D array and 1D array*) – A list containing the matrices C and c in the objective function above.
- **inequality_constraints** (*list containing a 2D array and 1D array*) – A list containing the matrices D and d in the objective function above.
- **allow_rank_deficient** (*bool*) – If True, allows the problem to be solved even if the design matrix is rank-deficient.

Returns
Tuple containing the optimized phase amounts (1D numpy.array), a covariance matrix corresponding to the optimized phase amounts (2D numpy.array), and the weighted residual of the fitting procedure (a float).

Return type
tuple

class burnman.optimize.nonlinear_fitting.**NonLinearModel**

Bases: ABC

Abstract base class that defines the required interface for models used in nonlinear least squares fitting. Models must support evaluation, normal vector computation, and parameter handling.

abstractmethod get_params()

Returns
Current model parameters as a NumPy array.

Return type
numpy.ndarray

abstractmethod set_params(*param_values*)

Parameters
param_values (*numpy.ndarray*) – Parameter values to set.

abstractmethod function(*x*, *flag=None*)

Parameters
- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Model output.

Return type
numpy.ndarray

abstractmethod normal(*x*, *flag=None*)

Parameters
- **x** (*numpy.ndarray*) – Input coordinates.
- **flag** (*any*) – Optional argument to control evaluation mode.

Returns
Normal vector.

Return type
numpy.ndarray

`validate()`

Ensure that all required model attributes are present before fitting. These include observed data, data covariance matrices, convergence tolerances for MLE projections, and finite difference step sizes.

Raises
`AssertionError` – If required attributes are missing.

burnman.optimize.nonlinear_fitting.**abs_line_project**(*M*, *n*)

Project a covariance matrix M onto a unit direction vector n. This gives the variance of the distribution in the direction n.

Parameters

- **M** (*numpy.ndarray*) – Covariance matrix.
- **n** (*numpy.ndarray*) – Direction vector.

Returns
Projected variance.

Return type
float

burnman.optimize.nonlinear_fitting.**mle_estimate**(*model*, *x*, *x_m*, *cov*, *flag*)

Find an approximation to the maximum likelihood estimate of a point on the model surface corresponding to noisy observation x, under Gaussian noise with covariance cov, given a starting estimate on the model surface x_m.

This approximation is the orthogonal projection of x onto the local tangent plane of the model surface, as determined using the Mahalanobis metric defined by cov.

Parameters

- **model** (NonLinearModel) – Model object.
- **x** (*numpy.ndarray*) – Observed data point.
- **x_m** (*numpy.ndarray*) – Initial guess on the model surface.
- **cov** (*numpy.ndarray*) – Covariance matrix.
- **flag** (*any*) – Optional flag for evaluation.

Returns
Tuple containing MLE position, residual, and projected variance.

Return type
tuple(numpy.ndarray, float, float)

burnman.optimize.nonlinear_fitting.**find_mle**(*model*)

Find the maximum likelihood point for each datum given the current model parameters.

This is done iteratively using the function mle_estimate.

Parameters
model (*FittableModel*) – Model object with required attributes.

Returns
MLE data positions, weighted residuals, and weights.

Return type
tuple(numpy.ndarray, numpy.ndarray, numpy.ndarray)

burnman.optimize.nonlinear_fitting.**calculate_jacobian**(*model*)

Compute the Jacobian matrix of weighted residuals with respect to model parameters using central finite differences.

Parameters
model (*FittableModel*) – Model object.

Modifies
model.jacobian — Populated with partial derivatives.

burnman.optimize.nonlinear_fitting.**nonlinear_least_squares_fit**(*model*, *lm_damping=0.0*, *param_tolerance=1e-07*, *max_lm_iterations=100*, *param_priors=None*, *param_prior_inv_cov_matrix=None*, *first_run_differential_evolution=False*, *differential_evolution_seed=42*, *verbose=False*)

Function to compute the "best-fit" parameters for a model by nonlinear least squares fitting.

The nonlinear least squares algorithm closely follows the logic in Section 23.1 of Bayesian Probability Theory (von der Linden et al., 2014; Cambridge University Press).

Parameters

- **model** (*FittableModel*) – Model with fitting interface.
- **lm_damping** (*float*) – Initial damping factor for the Levenberg-Marquardt updates. This is adapted automatically during fitting: increased whenever a step fails to reduce the objective function, and decreased whenever a step succeeds.
- **param_tolerance** (*float*) – Convergence tolerance based on fractional parameter change.
- **max_lm_iterations** (*int*) – Maximum number of LM iterations.
- **param_priors** (*1D numpy array*) – Prior values for the parameters.

- **param_prior_inv_cov_matrix** (*2D numpy array (square)*) – Inverse of the 1 sigma uncertainties for the prior values of the parameters.
- **verbose** (*bool*) – If True, print iteration status.

Modifies

model — Sets optimized parameters, covariance matrix, weighted residuals, Jacobian, and noise estimates.

Note

The object passed as model must have the following attributes:

- **data [2D numpy.array] - Elements of x[i][j] contain the** observed position of data point i.
- **data_covariances [3D numpy.array] Elements of cov[i][j][k] contain** the covariance matrix of data point i.
- **mle_tolerances [numpy.array] - The iterations to find the maximum likelihood** estimator for each observed data point will stop when mle_tolerances[i] < np.linalg.norm(data_mle[i] - model.function(data_mle[i], flag))
- delta_params [numpy.array] - parameter perturbations used to compute the jacobian

Must also have the following methods:

- set_params(self, param_values) - Function to set parameters.
- get_params(self) - Function to get current model parameters.
- function(self, x) - Returns value of model function evaluated at x.
- normal(self, x) - Returns value of normal to the model function evaluated at x.

After this function has been performed, the following attributes are added to model:

- dof [int] - Degrees of freedom of the system.
- **data_mle [2D numpy array] - Maximum likelihood estimates of the observed data points** on the best-fit curve.
- jacobian [2D numpy array] - d(weighted_residuals)/d(parameter).
- weighted_residuals [numpy array] - Weighted residuals.
- weights [numpy array] - 1/(data variances normal to the best fit curve).
- WSS [float] - Weighted sum of squares residuals.
- popt [numpy array] - Optimized parameters.
- pcov [2D numpy array] - Covariance matrix of optimized parameters.
- noise_variance [float] - Estimate of the variance of the data normal to the curve.

This function is available as `burnman.nonlinear_least_squares_fit`.

burnman.optimize.nonlinear_fitting.**nonlinear_least_squares_fit_differential_evolution**(*model*, *param_tole*
07,
param_pri
param_pri
pol-
ish=False,
seed=None

Function to compute the "best-fit" parameters for a model by nonlinear least squares fitting using differential evolution.

This function is intended to be used as a first step in the fitting process, especially when there might be multiple local minima and/or the initial parameter guesses are far from the optimal values. This function does not compute the covariance matrix of the optimized parameters. It is recommended to be followed by a call to *nonlinear_least_squares_fit* for further refinement.

Parameters

- **model** (*FittableModel*) – Model with fitting interface.
- **param_tolerance** (*float*) – Convergence tolerance, as used by scipy.optimize.differential_evolution.
- **param_priors** (*1D numpy array*) – Prior values for the parameters.
- **param_prior_inv_cov_matrix** (*2D numpy array (square)*) – Inverse of the 1 sigma uncertainties for the prior values of the parameters.
- **polish** (*bool*) – If True, perform a final local optimization using the Levenberg-Marquardt algorithm after differential evolution.
- **seed** (*int*) – The random seed to use for the differential evolution algorithm.

Modifies

model — Sets optimized parameters, covariance matrix, weighted residuals, Jacobian, and noise estimates.

Note

The object passed as model must have the following attributes: - *bounds*: A list of tuples specifying the lower and upper bounds for each parameter.

Other required attributes and methods are the same as those for *nonlinear_least_squares_fit*.

burnman.optimize.nonlinear_fitting.**confidence_prediction_bands**(*model*, *x_array*, *confidence_interval*, *f*, *flag=None*)

This function calculates the confidence and prediction bands of the function f(x) from a best-fit model with uncertainties in its parameters as calculated (for example) by the function nonlinear_least_squares_fit().

The values are calculated via the delta method, which estimates the variance of f evaluated at x as var(f(x)) = df(x)/dB var(B) df(x)/dB where df(x)/dB is the vector of partial derivatives of f(x) with respect to B.

Parameters

- **model** (*object*) – As modified (for example) by the function burnman.nonlinear_least_squares_fit(). Should contain the following functions: get_params, set_params, function, normal And attributes: delta_params, pcov, dof, noise_variance
- **x_array** (*2D numpy.array*) – Coordinates at which to evaluate the bounds.
- **confidence_interval** (*float*) – Probability level of finding the true model (confidence bound) or any new data point (probability bound). For example, the 95% confidence bounds should be calculated using a confidence interval of 0.95.
- **f** (*function*) – The function defining the variable y=f(x) for which the confidence and prediction bounds are desired.
- **flag** (*type informed by model object*) – This (optional) flag is passed to model.function to control how the modified position of x is calculated. This value is then used by f(x)

Returns

An element of bounds[i][j] gives the lower and upper confidence (i=0, i=1) and prediction (i=2, i=3) bounds for the jth data point.

Return type

2D numpy.array

burnman.optimize.nonlinear_fitting.**plot_cov_ellipse**(*cov*, *pos*, *nstd=2*, *ax=None*, ***kwargs*)

Plots an *nstd* sigma error ellipse based on the specified covariance matrix (*cov*). Additional keyword arguments are passed on to the ellipse patch artist.

Parameters

- **cov** (*numpy.array*) – The 2x2 covariance matrix to base the ellipse on.
- **pos** (*list or numpy.array*) – The location of the center of the ellipse. Expects a 2-element sequence of [x0, y0].
- **nstd** (*float*) – The radius of the ellipse in numbers of standard deviations. Defaults to 2 standard deviations.
- **ax** (*matplotlib.pyplot.axes*) – The axis that the ellipse will be plotted on. Defaults to the current axis.
- **kwargs** – Additional keyword arguments are passed on to the ellipse patch.

Returns

The covariance ellipse (already applied to the desired axes object).

Return type

matplotlib.patches.Ellipse

burnman.optimize.nonlinear_fitting.**corner_plot**(*popt*, *pcov*, *param_names=[]*, *n_std=1.0*)

Creates a corner plot of covariances

Parameters

- **popt** (*numpy.array*) – Optimized parameters.
- **pcov** (*2D numpy.array*) – Covariance matrix of the parameters.
- **param_names** (*list*) – Parameter names.
- **n_std** (*float*) – Number of standard deviations for ellipse.

Returns

matplotlib.pyplot.figure and list of matplotlib.pyplot.Axes objects.

Return type

tuple

burnman.optimize.nonlinear_fitting.**weighted_residual_plot**(*ax, model, flag=None, sd_limit=3, cmap=<matplotlib.colors.LinearSegmentedCol object>, plot_axes=[0, 1], scale_axes=[1.0, 1.0]*)

Creates a plot of the weighted residuals The user can choose the projection axes, and scaling to apply to those axes The chosen color palette (cmap) is discretised by standard deviation up to a cut off value of sd_limit.

Parameters

- **ax** – Plot.
- **type** – matplotlib.pyplot.Axes
- **model** (*object*) – A model as used by burnman.nonlinear_least_squares_fit(). Must contain the attributes model.data, model.weighted_residuals and model.flags (if flag is not None).
- **flag** (*str*) – String to determine which data to plot. Finds matches with model.flags.
- **sd_limit** (*float*) – Data with weighted residuals exceeding this limit are plotted in black.
- **cmap** (*matplotlib color palette*) – Color palette.
- **plot_axes** (*list of int*) – Data axes to use as plot axes.
- **scale_axes** (*list of float*) – Plot axes are scaled by multiplication of the data by these values.

Returns

Coloured scatter plot of the weighted residuals in data space.

Return type

matplotlib Axes object

burnman.optimize.nonlinear_fitting.**extreme_values**(*weighted_residuals*, *confidence_interval*)

This function uses extreme value theory to calculate the number of standard deviations away from the mean at which we should expect to bracket *all* of our n data points at a certain confidence level.

It then uses that value to identify which (if any) of the data points lie outside that region, and calculates the corresponding probabilities of finding a data point at least that many standard deviations away.

Parameters

- **weighted_residuals** (*array of float*) – Array of residuals weighted by the square root of their variances wr_i = r_i/sqrt(var_i).
- **confidence_interval** (*float*) – Probability at which all the weighted residuals lie within the confidence bounds.

Returns

Number of standard deviations at which we should expect to encompass all data at the user-defined confidence interval, the indices of weighted residuals exceeding the confidence_interval defined by the user, and the probabilities that the extreme data point of the distribution lies further from the mean than the observed position wr_i for each i in the "indices" output array.

Return type

tuple of (float, numpy.array, numpy.array)

burnman.optimize.nonlinear_fitting.**plot_residuals**(*ax*, *weighted_residuals*, *n_bins=None*, *flags=[]*)

class burnman.optimize.nonlinear_solvers.**Iterates**(*x: list = <factory>*, *F: list = <factory>*, *lmda: list = <factory>*)

Bases: object

Container for storing iteration history of the solver. Attributes are lists of the iterates x, function evaluations F, and step lengths relative to the full Newton step lmda.

x: list

F: list

lmda: list

append(*x*, *F*, *lmda*)

Append an iteration's data to the history. :param x: Current iterate. :param F: Current function evaluation. :param lmda: Current step length.

close()

Finalize the iteration history by converting lists to arrays.

class burnman.optimize.nonlinear_solvers.**TerminationCode**(**values*)

Bases: Enum

Enumeration of termination codes for the nonlinear solver. These are used to indicate the reason for solver termination. The codes are:

- SUCCESS (0): The solver successfully found a root.
- TOO_NONLINEAR (1): The function is too non-linear for lower lambda bound.
- CONSTRAINT_VIOLATION (2): The descent vector crosses the constraints.
- MAX_ITERATIONS (3): The solver reached the maximum number of iterations.
- SINGULAR_FAIL (4): The system was effectively singular and failed to converge.
- SINGULAR_SUCCESS (5): The solver found a root but the system is singular.

SUCCESS = 0

TOO_NONLINEAR = 1

CONSTRAINT_VIOLATION = 2

MAX_ITERATIONS = 3

SINGULAR_FAIL = 4

SINGULAR_SUCCESS = 5

class burnman.optimize.nonlinear_solvers.**Solution**(*x: ndarray | None = None*, *n_it: int = 0*, *F: ndarray | None = None*, *F_norm: float | None = None*, *J: ndarray | None = None*, *code: TerminationCode | None = None*, *text: str | None = None*, *success: bool = False*, *store_iterates: bool = False*)

Bases: `object`

Container for the solution of a nonlinear system of equations.

Parameters

- **x** (*np.ndarray, optional*) – Final solution vector.
- **n_it** (*int, optional, default is 0*) – Number of iterations performed.
- **F** (*np.ndarray, optional*) – Function evaluation at the solution.
- **F_norm** (*float, optional*) – Euclidean norm of F at the solution.
- **J** (*np.ndarray, optional*) – Jacobian matrix at the solution.
- **code** (*TerminationCode, optional*) – Termination code. Valid codes and their values are given by the TerminationCode enum.
- **text** (*str, optional*) – Human-readable termination description.
- **success** (*bool, optional, default is False*) – True if the solver converged.
- **store_iterates** (*bool, optional, default is False*) – If True, iteration history is stored in an *iterates* attribute. The Iterates instance contains lists of all

iterates x, function evaluations F, and step lengths relative to the full Newton step lmda.

x: ndarray | None = None

n_it: int = 0

F: ndarray | None = None

F_norm: float | None = None

J: ndarray | None = None

code: *TerminationCode* | None = None

text: str | None = None

success: bool = False

store_iterates: bool = False

terminate(*converged: bool*, *minimum_lmda: bool*, *persistent_bound_violation: bool*, *lmda_bounds: tuple[float, float]*, *n_it: int*, *max_iterations: int*, *violated_constraints: list[int]*, *singular_jacobian: bool*, *condition_number: float*) → None

Set the termination status of the solver.

class burnman.optimize.nonlinear_solvers.**DampedNewtonSolver**(*F*, *J*, *guess*, *tol: float | NDArray = 1e-06*, *F_tol: float | NDArray = 1e-08*, *max_iterations: int = 100*, *lambda_bounds=<function DampedNewtonSolver.<lambda>>*, *linear_constraints=(0.0, array([-1.]))*, *store_iterates: bool = False*, *regularization: float = np.float64(2.220446049250313e-16)*, *cond_lu_thresh: float = 1000000000000.0*, *constraint_thresh: float = np.float64(4.440892098500626e-16)*)

Bases: object

Solver for the multivariate nonlinear system F(x) = 0 with Jacobian J(x), using the damped affine invariant modification to Newton's method (Deuflhard, 1974; 1975; 2004).

This implementation follows the algorithm described in Nowak and Weimann (1991): Technical Report TR-91-10, Algorithm B, modified to accept linear inequality constraints.

Linear inequality constraints are provided by a tuple argument named linear_constraints. The constraints are satisfied if A*x + b <= 0. Note that the sign of b is opposite to that used in

scipy.optimize.LinearConstraint. If any constraints are violated by the current candidate solution using step size lambda, lambda is reduced to satisfy all constraints.

If the current iterate lies on one or more constraints and the Newton step violates one or more of those constraints, the next step is calculated via the method of Lagrangian multipliers, minimizing the L2-norm of F(x_{i+1}) subject to the violated constraints.

Successful termination of the solver requires meeting all of the following criteria:

- all(np.abs(dx (simplified Newton step)) < tol)
- **all(np.abs(dx (full Newton step)) < np.sqrt(10*tol))**
  (to avoid pathological cases)
- lambda = lambda_bounds(dx, x)[1] (lambda = 1 for a full Newton step)

If these criteria are not satisfied, iterations continue until one of the following occurs:

- lambda is reduced to its minimum value (indicating a very nonlinear problem)
- successive iterations have descent vectors which violate the constraints
- the maximum number of iterations is reached

solve() → *Solution*

Execute the damped Newton solver to find a root of F(x) = 0, optionally subject to linear inequality constraints A.x + b <= 0.

If the solver fails to converge, it terminates with an informative status code and description.

Returns
Solution object. See *Solution* class for details of its attributes.

Return type
Solution instance

burnman.optimize.nonlinear_solvers.**damped_newton_solve**(*F*, *J*, *guess*, *tol: float | NDArray = 1e-06*, *F_tol: float | NDArray = 1e-08*, *max_iterations: int = 100*, *lambda_bounds=<function <lambda>>*, *linear_constraints=(0.0, array([-1.]))*, *store_iterates: bool = False*) → *Solution*

Helper function to run the DampedNewtonSolver.

Parameters

- **F** (*callable[[np.ndarray], np.ndarray]*) – Function that evaluates the system of nonlinear equations F(x) = 0 at a given x.
- **J** (*callable[[np.ndarray], np.ndarray]*) – Function that evaluates the Jacobian matrix J(x) of F(x) at a given x.
- **guess** (*np.ndarray*) – Initial guess for the solution vector x.

- **tol** (*float, optional*) – Convergence tolerance for Newton iterations, defaults to 1.0e-6.
- **max_iterations** (*int, optional*) – Maximum number of Newton iterations to perform, defaults to 100.
- **lambda_bounds** (*callable[[np.ndarray, np.ndarray], tuple[float, float]], optional*) – Function returning (min_lambda, max_lambda) for the damping factor given the current Newton step dx and point x, defaults to *lambda dx, x: (1.0e-8, 1.0)*.
- **linear_constraints** (*tuple[np.ndarray, np.ndarray], optional*) – Tuple (A, b) representing linear inequality constraints A.x + b <= 0, defaults to (0.0, np.array([-1.0])).
- **store_iterates** (*bool, optional*) – If True, stores all intermediate iterates, function evaluations, and lambda values in the solution object, defaults to False.

Returns
Solution object. See *Solution* class for details of its attributes.

Return type
Solution instance

20.23 Tools

Burnman has a number of high-level tools to help achieve common goals. Several of these have already been described in previous sections.

20.23.1 Unit cell

20.23.2 Equilibration

burnman.tools.equilibration.**calculate_constraints**(*assemblage*, *n_free_compositional_vectors*)

This function calculates the linear inequality constraints bounding the valid parameter space for a given assemblage.

The constraints are as follows:

- Pressure and temperature must be positive
- All phase fractions must be positive
- All site-species occupancies must be positive

The constraints are stored in a vector (b) and matrix (A). The sign convention is chosen such that the constraint is satisfied if A.x + b < eps.

Parameters
assemblage (*burnman.Composite*) – The assemblage for which the constraints are calculated.

Returns
The constraints vector and matrix.

Return type
tuple

burnman.tools.equilibration.**get_parameters**(*assemblage*, *n_free_compositional_vectors=0*)

Gets the starting parameters vector (x) for the current equilibrium problem. These are:

- pressure
- temperature
- absolute amount of each phase. if a phase is a solution with >1 endmember, the following parameters are the mole fractions of the independent endmembers in the solution, except for the first endmember (as the mole fractions must sum to one).

Parameters
assemblage (*burnman.Composite*) – The assemblage to be equilibrated.

Returns
The current values of all the parameters.

Return type
numpy.array

burnman.tools.equilibration.**get_endmember_amounts**(*assemblage*)

Gets the absolute amounts of all the endmembers in the solution.

Parameters
assemblage (*burnman.Composite*) – The assemblage to be equilibrated.

Returns
The current amounts of all the endmembers.

Return type
numpy.array

burnman.tools.equilibration.**set_compositions_and_state_from_parameters**(*assemblage*, *parameters*)

Sets the phase compositions, amounts and state of the assemblage from a list of parameter values.

Parameters

- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **parameters** (*numpy.array*) – The current parameter values.

burnman.tools.equilibration.**default_F_tolerances**(*assemblage*, *equality_constraints*, *n_atoms*)

Returns the default tolerances for each component of F(x).

burnman.tools.equilibration.**F**(*x*, *assemblage*, *equality_constraints*, *reduced_composition_vector*, *reduced_free_composition_vectors*)

The vector-valued function for which the root is sought. The first two vector values depend on the equality_constraints chosen. For example, if

- eq[i][0] = 'P', F[i] = P - eq[i][1]
- eq[i][0] = 'T', F[i] = T - eq[i][1]
- eq[i][0] = 'S', F[i] = entropy - eq[i][1]
- eq[i][0] = 'V', F[i] = volume - eq[i][1]
- eq[i][0] = 'X', np.dot(eq[i][1][0], x) - eq[i][1][1]

The next set of vector values correspond to the reaction affinities. The final set of vector values correspond to the bulk composition constraints.

Parameters

- **x** (*numpy array*) – Parameter values for the equilibrium problem to be solved.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of lists*) – A list of the equality constraints (see above for valid formats).
- **reduced_composition_vector** (*numpy.array*) – The amounts of the independent elements.
- **reduced_free_composition_vectors** (*2D numpy.array*) – The amounts of the independent elements in each of the free_compositional_vectors.

Returns

The vector corresponding to F(x).

Return type

numpy.array

burnman.tools.equilibration.**jacobian**(*x*, *assemblage*, *equality_constraints*, *reduced_free_composition_vectors*)

The Jacobian of the vector-valued function F for which the root is sought ($\partial F/\partial x$). See documentation for *F()* and *get_parameters()* (which return F and x respectively) for more details.

Parameters

- **x** (*numpy.array*) – Parameter values for the equilibrium problem to be solved.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of lists*) – A list of the equality constraints (see documentation for burnman.tools.equilbration.F()).
- **reduced_free_composition_vectors** (*2D numpy array*) – The amounts of the independent elements in each of the free_compositional_vectors.

Returns

The Jacobian for the equilibrium problem.

Return type
2D numpy.array

burnman.tools.equilibration.**lambda_bounds**(*dx*, *x*, *endmembers_per_phase*)

Returns the lambda bounds for the damped affine invariant modification to Newton's method for non-linear problems (Deuflhard, 1974;1975;2004).

Parameters

- **dx** (*numpy.array*) – The proposed newton step.
- **x** (*numpy.array*) – Parameter values for the equilibrium problem to be solved.
- **endmembers_per_phase** (*list of int*) – A list of the number of endmembers in each phase.

Returns
Minimum and maximum allowed fractions of the full newton step (dx).

Return type
tuple of floats

burnman.tools.equilibration.**phase_fraction_constraints**(*phase*, *assemblage*, *fractions*, *prm*)

Converts a phase fraction constraint into standard linear form that can be processed by the root finding problem.

We start with a single fraction or an array of fractions for a particular phase ($n_p / \sum n = f$). These are then converted into the "X" form of constraint by multiplying by $\sum n$ and moving all terms to the LHS of the equation:

$-fn_0 - fn_1 - \ldots + (1-f)n_p - \ldots = 0$

This form is less readable, but easier to use as a constraint in a nonlinear solve.

Parameters

- **phase** (*burnman.Solution* or *burnman.Mineral*) – The phase for which the fraction is to be constrained
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **fractions** (*numpy.array*) – The phase fractions to be satified at equilibrium.
- **prm** (*namedtuple*) – A tuple with attributes n_parameters (the number of parameters for the current equilibrium problem) and phase_amount_indices (the indices of the parameters that correspond to phase amounts).

Returns
The phase fraction constraints.

Return type
list

burnman.tools.equilibration.**phase_composition_constraints**(*phase*, *assemblage*, *constraints*, *prm*)

Converts a phase composition constraint into standard linear form that can be processed by the root finding problem.

We start with constraints in the form (site_names, n, d, v), where $(nx)/(dx) = v$ and n and d are fixed vectors of site coefficients. So, one could for example choose a constraint ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) which would correspond to equal amounts Mg and Fe on the A site.

This function converts the user-defined vectors of site constraints n and d into vectors of endmember proportion constraints n' and d', such that $(n'x)/(d'x) = v$. This is done via linear transformation using the site occupancy matrix provided by *burnman.Solution*. By multiplying by the denominator, we have the following scalar comparison: $(n'x) = v(d'x)$

The equilibration function does not use the proportion of the first endmember (as the endmember proportions must sum to one), and so we split x, n' and d' into the first element and following elements: $(n'_0 x_0 + n'_i x_i) = v(d'_0 x_0 + d'_i x_i)$ where i is taken over all elements apart from the first.

With some more rearranging we can express the constraint in standard linear form: $(n'_0(1 - \sum_j x_j) + n'_i x_i) = v(d'_0(1 - \sum_j x_j) + d'_i x_i)$

$(n'_0 + (n'_i - 1_i n'_0)x_i) = v(d'_0 + (d'_i - 1_i d'_0)x_i)$

$(((n'_i - 1_i n'_0) - v(d'_i - 1_i d'_0))x_i) = (vd'_0 - n'_0)$

This form is less readable, but easier to use as a constraint in a nonlinear solve.

Parameters

- **phase** (*burnman.Solution*) – The phase for which the composition is to be constrained.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **constraints** (*tuple*) – The desired constraints in the form: site_names (list of strings), numerator (numpy.array), denominator (numpy.array), values (numpy.array).

Returns

The phase composition constraints in standard form.

Return type

list

burnman.tools.equilibration.**get_equilibration_parameters**(*assemblage*, *composition*, *free_compositional_vectors*)

Builds a named tuple containing the parameter names and various other parameters needed by the equilibrium solve.

Parameters

- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **composition** (*dict*) – The bulk composition for the equilibrium problem.
- **free_compositional_vectors** (*list of dictionaries*) – The bulk compositional degrees of freedom for the equilibrium problem.

Returns

A tuple with attributes:

- **parameter_names: list of strings, the names of the parameters**
  for the current equilibrium problem
- default_tolerances: numpy.array, the default tolerances for each parameter
- n_parameters: int, the number of parameters for the current equilibrium problem
- **phase_amount_indices: list of int, the indices of the parameters that**
  correspond to phase amounts
- bulk_composition_vector: numpy.array, the bulk composition vector
- free_compositional_vectors: 2D numpy.array, the free compositional vectors
- **reduced_composition_vector: numpy.array, the bulk composition vector**
  reduced to the independent elements
- **reduced_free_composition_vectors: 2D numpy.array, the free compositional vectors**
  reduced to the independent elements
- **constraint_vector: numpy.array, vector for the linear constraints bounding the**
  valid parameter space (b in Ax + b < eps)
- **constraint_matrix: 2D numpy.array, matrix for the linear constraints bounding the**
  valid parameter space (A in Ax + b < eps)

Return type
namedtuple

burnman.tools.equilibration.**process_eq_constraints**(*equality_constraints*, *assemblage*, *prm*)

A function that processes the equality constraints into a form that can be processed by the F and jacobian functions.

This function has two main tasks: it turns phase_fraction and phase_composition constraints into standard linear constraints in the solution parameters. It also turns vector-valued constraints into a list of scalar-valued constraints.

Parameters

- **equality_constraints** (*list*) – A list of equality constraints. For valid types of constraints, please see the documentation for `burnman.equilibrate()`.
- **assemblage** (*burnman.Composite*) – The assemblage to be equilibrated.
- **prm** (*namedtuple*) – A tuple with attributes n_parameters (the number of parameters for the current equilibrium problem) and phase_amount_indices (the indices of the parameters that correspond to phase amounts).

Returns
Equality constraints in a form that can be processed by the F and jacobian functions.

Return type
list of lists

burnman.tools.equilibration.**equilibrate**(*composition*, *assemblage*, *equality_constraints*, *free_compositional_vectors=[]*, *tol=None*, *store_iterates=False*, *store_assemblage=True*, *max_iterations=100.0*, *verbose=False*)

A function that finds the thermodynamic equilibrium state of an assemblage subject to given equality constraints by solving a set of nonlinear equations related to the chemical potentials and other state variables of the system.

The user chooses an assemblage (e.g. olivine, garnet and orthopyroxene) and $2 + n_c$ equality constraints, where n_c is the number of bulk compositional degrees of freedom. The equilibrate function attempts to find the remaining unknowns that satisfy those constraints.

There are a number of equality constraints implemented in burnman. These are given as a list of lists. Each constraint should have the form: [<constraint type>, <constraint>], where <constraint type> is one of 'P', 'T', 'S', 'V', 'X', 'phase_fraction', or 'phase_composition'. The format of the <constraint> object depends on the constraint type:

- **P: float or numpy.array of**
  pressures [Pa]
- **T: float or numpy.array of**
  temperatures [K]
- **S: float or numpy.array of**
  entropies [J/K]
- **V: float or numpy.array of**
  volumes [m:math:^3]
- **phase_fraction: tuple with the form (<phase> <fraction(s)>),**
  where <phase> is one of the phase objects in the assemblage and <fraction(s)> is a float or numpy.array corresponding to the desired phase fractions.
- **phase_composition: tuple with the form (<phase> <constraint>),**
  where <phase> is one of the phase objects in the assemblage and <constraint> has the form (site_names, n, d, v), where $(nx)/(dx) = v$, n and d are constant vectors of site coefficients, and v is a float or numpy.array. For example, a constraint of the form ([Mg_A, Fe_A], [1., 0.], [1., 1.], [0.5]) would correspond to equal amounts Mg and Fe on the A site (Mg_A / (Mg_A + Fe_A) = 0.5).
- **X: list of a numpy.array and a float or numpy.array,**
  where the equality satisfies the linear equation arr[0] x = arr[1]. The first numpy.array is fixed, and the second is to be looped over by the equilibrate function. This is a generic compositional equality constraint.

Parameters

- **composition** (*dict*) – The bulk composition that the assemblage must satisfy.

- **assemblage** (*burnman.Composite* or *burnman.RelaxedComposite*) – The assemblage to be equilibrated.
- **equality_constraints** (*list of list*) – The list of equality constraints. See above for valid formats.
- **free_compositional_vectors** (*list of dict*) – A list of dictionaries containing the compositional freedom of the solution. For example, if the list contains the vector {'Mg': 1., 'Fe': -1}, that implies that the bulk composition is equal to composition + a (n_Mg - n_Fe), where the value of a is to be determined by the solve. Vector given in atomic (molar) units of elements.
- **tol** (*float or numpy.array*) – The tolerance for the nonlinear solver. This can be a single float, which is then applied to all parameters, or a numpy array with the same length as the number of parameters in the solve (2 + number of phases + number of free compositional vectors). The default is None, which sets the tolerances to 1 Pa for pressure, 1e-3 K for temperature, 1e-6 for phase amounts and 1e-6 for free compositional vectors. One of the termination criteria for the nonlinear solver is that the absolute change in each parameter is less than the corresponding tolerance.
- **store_iterates** (*bool*) – Whether to store the parameter values for each iteration in each solution object.
- **store_assemblage** (*bool*) – Whether to store a copy of the assemblage object in each solution object.
- **max_iterations** (*int*) – The maximum number of iterations for the nonlinear solver.
- **verbose** (*bool*) – Whether to print output updating the user on the status of equilibration.

Returns

Solver solution object (or a list, or a 2D list of solution objects) created by *burnman.optimize.nonlinear_solvers.damped_newton_solve()*, and a namedtuple object created by *burnman.tools.equilibration.get_equilibration_parameters()*. See documentation of these functions for the return types. If store_assemblage is True, each solution object also has an attribute called *assemblage*, which contains a copy of the input assemblage with the equilibrated properties. So, for a 2D grid of solution objects, one could call either sols[0][1].x[0] or sols[0][1].assemblage.pressure to get the pressure.

Return type

tuple

20.23.3 Optimal thermobarometry

burnman.tools.thermobarometry.**get_reaction_matrix**(*assemblage*, *small_fraction_tol=0.0*)

Set up a matrix of independent reactions for an assemblage, possibly reducing the number of endmembers by excluding components with small molar fractions.

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to build the reaction matrix.
- **small_fraction_tol** (*float, optional, default 0.0*) – If > 0.0, reduces the number of endmembers in solution phases by transforming to a smaller set of independent endmembers using a greedy algorithm and excluding those with molar fractions smaller than this value during the inversion.

Returns
The reaction matrix for the assemblage, returned in the original endmember basis.

Return type
2D np.array

burnman.tools.thermobarometry.**assemblage_set_state_from_params**(*assemblage*, *params*)

Set the state of the assemblage (P, T, compositions) from the given list of parameters.

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to set the state. If there are free compositional vectors for any solution phases (e.g., to account for an unknown state of order at fixed composition), the corresponding phases must have the attributes *baseline_composition* and *free_compositional_vectors* set, and the length of params must match the number of free compositional vectors plus two (for P and T).
- **params** (*list or np.array*) – List of parameters, where the first two are pressure (Pa) and temperature (K), and any additional parameters correspond to compositional degrees of freedom. Each compositional degree of freedom corresponds to a free compositional vector in one of the solution phases in the assemblage.

Returns
None

Return type
None

burnman.tools.thermobarometry.**assemblage_affinity_covariance_matrix**(*assemblage*, *reaction_matrix*, *dataset_covariances=None*, *include_state_uncertainties=False*)

Compute the covariance matrix of the affinities of the independent reactions for the given assemblage. This can include contributions from compositional uncertainties, dataset uncertainties, and from uncertainties in pressure and temperature.

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to compute the covariance matrix. Should have its state (P, T, compositions) set prior to calling this function.
- **reaction_matrix** (*2D np.array*) – The reaction matrix for the assemblage.

- **dataset_covariances** (*dict, with keys 'endmember_names' and 'covariance_matrix'. Default is None, in which case only compositional uncertainties are considered.*) – The covariance data from the thermodynamic dataset.
- **include_state_uncertainties** (*bool*) – If True, includes the contribution from uncertainties in pressure and temperature to the covariance matrix. If True, the assemblage must have the attribute *state_covariances*.

Returns
Covariance matrix of the affinities of the independent reactions.

Return type
2D np.array

burnman.tools.thermobarometry.**assemblage_affinity_misfit**(*assemblage*, *reaction_matrix*, *dataset_covariances=None*, *include_state_uncertainties=False*)

Compute the objective misfit function (chi-squared) for given P and T.

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to compute the misfit. Should have its state (P, T, compositions) set prior to calling this function.
- **reaction_matrix** (*2D np.array*) – The reaction matrix for the assemblage.
- **dataset_covariances** (*dict, with keys 'endmember_names' and 'covariance_matrix'. Default is None, in which case only compositional uncertainties are considered.*) – The covariance data from the thermodynamic dataset.
- **include_state_uncertainties** (*bool*) – If True, includes the contribution from uncertainties in pressure and temperature to the covariance matrix. If True, the assemblage must have the attribute *state_covariances*.

Returns
Chi-squared misfit value.

Return type
float

burnman.tools.thermobarometry.**assemblage_state_misfit**(*assemblage*)

Compute the objective misfit function (chi-squared) for given P and T based on prior expectations for P and T and their covariance. The assemblage must have attributes *state_priors* and *state_inverse_covariances*.

Parameters
assemblage (*Assemblage*) – The mineral assemblage for which to compute the misfit.

Returns
Chi-squared misfit value.

Return type
float

burnman.tools.thermobarometry.**estimate_conditions**(*assemblage*, *dataset_covariances=None*, *include_state_misfit=False*, *guessed_conditions=[1000000000.0, 1000.0]*, *pressure_bounds=[100000.0, 400000000000.0]*, *temperature_bounds=[300.0, 4000.0]*, *P_scaling=1000000.0*, *small_fraction_tol=0.0*, *max_it=100*)

Perform a least-squares inversion to find the optimal pressure and temperature for a given mineral assemblage. Algorithm modified from [PH94].

Parameters

- **assemblage** (*Assemblage*) – The mineral assemblage for which to perform the inversion. Each solution phase in the assemblage must have its composition set along with its compositional covariance matrix (called *compositional_covariances*). If there are compositional degrees of freedom, they can be added by setting the attribute *free_compositional_vectors* on the relevant solution phases, where each row of the array corresponds to a free compositional vector, and the columns correspond to the amounts of endmembers of that phase in each vector.
- **dataset_covariances** (*dict, with keys 'endmember_names' and 'covariance_matrix'. Default is None, in which case only compositional uncertainties are considered.*) – The covariance data describing the uncertainties in endmember energies, taken from the thermodynamic dataset. For example, the covariance for the Holland et al. (2018) dataset can be obtained using `burnman.minerals.HGP_2018_ds633.cov()`.
- **include_state_misfit** (*bool*) – If True, includes the misfit from prior expectations on P and T. The assemblage must also have attributes *state_priors* and *state_inverse_covariances*.
- **guessed_conditions** (*np.array of shape (2,), optional, default None*) – Initial guess for pressure (Pa) and temperature (K). If not provided, the initial guess will be taken from the current state of the assemblage.
- **pressure_bounds** (*list of two floats*) – Bounds for pressure (Pa) during optimization.
- **temperature_bounds** (*list of two floats*) – Bounds for temperature (K) during optimization.
- **P_scaling** (*float*) – Scaling factor for pressure to improve numerical stability.
- **small_fraction_tol** (*float, optional, default 0.0*) – If > 0.0, reduces the number of endmembers in solution phases by transforming to a smaller set of independent endmembers using a greedy algorithm and excluding those with molar fractions smaller than this value during the inversion.

- **max_it** (*int, optional, default 100*) – Maximum number of iterations for the optimization algorithm.

Returns

Result object from the optimization containing the optimal conditions (x, which includes P, T, and any free compositional vectors) and other properties of the solution. These include the covariance matrix of the estimated parameters (xcov), the standard deviations of the estimated parameters (var), the correlation matrix (xcorr), the affinities of the independent reactions at the optimal conditions (affinities), the covariance matrix of the affinities (acov), the affinities weighted by the inverse square root of their covariance matrix (weighted_affinities), the partial derivatives of the affinities with respect to P and T (dadx), the number of independent reactions (n_reactions), the number of fitted parameters (n_params), the degrees of freedom (degrees_of_freedom), the chi-squared value (fun), the reduced chi-squared value (reduced_chisqr, given by the reduced chi-squared divided by the number of degrees of freedom), and the fit quality (fit, given by the square root of the reduced chi-squared value).

Return type

OptimizeResult

20.23.4 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*, *equilibration_function=None*)

Checks that numerical derivatives of the Gibbs energy of a mineral or composite under given conditions are equal to those provided analytically by the equation of state.

Parameters

- **m** (*burnman.Mineral* or *burnman.Composite*) – The material for which the equation of state is to be checked for consistency.
- **P** (*float*) – The pressure at which to check consistency.
- **T** (*float*) – The temperature at which to check consistency.
- **tol** (*float*) – The fractional tolerance for each of the checks.
- **verbose** (*bool*) – Decide whether to print information about each check.
- **including_shear_properties** (*bool*) – Decide whether to check shear information, which is pointless for liquids and equations of state without shear modulus parameterizations.
- **equilibration_function** (*function*) – Function to internally equilibrate object. Called after every set_state. Takes the mineral object as its only argument.

Returns

Boolean stating whether all checks have passed.

Return type

bool

burnman.tools.eos.**check_anisotropic_eos_consistency**(*m*, *P=1000000000.0*, *T=300.0*, *tol=0.0001*, *verbose=False*, *equilibration_function=None*)

Checks that numerical derivatives of the Gibbs energy of an anisotropic mineral under given conditions are equal to those provided analytically by the equation of state.

Parameters

- **m** (*burnman.AnisotropicMineral*) – The anisotropic mineral for which the equation of state is to be checked for consistency.
- **P** (*float*) – The pressure at which to check consistency.
- **T** (*float*) – The temperature at which to check consistency.
- **tol** (*float*) – The fractional tolerance for each of the checks.
- **verbose** (*bool*) – Decide whether to print information about each check.
- **equilibration_function** (*function*) – Function to internally equilibrate object. Called after every set_state. Takes the mineral object as its only argument.

Returns

Boolean stating whether all checks have passed.

Return type

bool

burnman.tools.eos.**check_hyperelastic_eos_consistency**(*m*, *deformation_gradient=array([[0.9, 0.1, 0.05], [0.1, 0.91, 0.04], [0.02, 0.03, 0.89]])*, *T=300.0*, *tol=0.0001*, *verbose=False*, ** (Keyword-only parameters separator (PEP 3102))*, *strain_step=0.0002*, *temperature_step=0.5*)

Check Helmholtz derivatives, Maxwell relations and frame indifference.

The checked cell matrix is `M = deformation_gradient @ M_reference`, where cell vectors are rows of `M.T`. All stresses and tangents are compared in the same spatial frame. Crystal-frame output is transformed back into the spatial frame at each perturbed state; the selected output frame is left unchanged. Matrix comparisons use Voigt notation with engineering shear strains (2 epsilon_23, 2 epsilon_13, 2 epsilon_12) and unscaled shear stresses.

The molar Helmholtz differential is $dF = -SdT + V\sigma : d\epsilon$. Consequently, $\pi = (\partial\sigma/\partial T)_M = -(1/V)(\partial S/\partial\epsilon)_T$ and $C_S = C_T + VT\pi \otimes \pi/C_\epsilon$. The spatial stiffness and compliance tensors need not have major symmetry.

An existing cell/temperature state is restored, even if exceptions are raised. An initially unset mineral is left at the checked state.

Parameters

- **m** – Hyperelastic mineral to check.

- **deformation_gradient** – Finite (3, 3) matrix with positive determinant, declared relative to `m.hydrostatic.cell_vectors_0`.
- **T** – Temperature in Kelvin.
- **tol** – Fractional tolerance, relative to the largest absolute entry in each compared scalar or tensor. Entropy and energy comparisons use C_eps and max(T C_eps, V max(abs(C_T))) as characteristic scales, so zero reference values do not require zero numerical error.
- **verbose** – Print each result and its maximum absolute error.
- **strain_step** – Positive dimensionless central-difference half-step.
- **temperature_step** – Positive temperature half-step in Kelvin, less than T. The perturbed states must be within the model's valid range.

Returns
Whether every check passes.

Return type
bool

20.23.5 Solution

burnman.tools.solution.**transform_solution_to_new_basis**(*solution*, *new_basis*, *n_mbrs=None*, *solution_name=None*, *endmember_names=None*, *molar_fractions=None*)

Transforms a solution model from one endmember basis to another. Returns a new Solution object.

Parameters

- **solution** (*burnman.Solution* object) – The original solution object.
- **new_basis** (*2D numpy array*) – The new endmember basis, given as amounts of the old endmembers.
- **n_mbrs** (*float, optional*) – The number of endmembers in the new solution (defaults to the length of new_basis).
- **solution_name** (*str, optional*) – A name corresponding to the new solution.
- **endmember_names** (*list of str, optional*) – A list corresponding to the names of the new endmembers.
- **molar_fractions** (*numpy.array, optional*) – Fractions of the new endmembers in the new solution.

Returns
The transformed solution.

Return type
burnman.Solution object

20.23.6 Chemistry

burnman.tools.chemistry.**fugacity**(*standard_material*, *assemblage*)

Calculates the fugacity of a standard material in another assemblage.

> **Note**
>
> set_method and set_state should already have been used on both assemblages.

Parameters

- **standard_material** – Standard material for which to calculate the fugacity. The material must have a formula as a dictionary parameter.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.

Returns

Value of the fugacity of the component with respect to the standard material.

Return type

float

burnman.tools.chemistry.**relative_fugacity**(*component_formula*, *assemblage*, *reference_assemblage*)

Calculates the fugacity of a chemical component in one assemblage relative to another one.

> **Note**
>
> set_method and set_state should already have been used on both assemblages.

Parameters

- **component_formula** (*dictionary*) – Chemical formula for which to compute the relative fugacity.
- **assemblage** (*burnman.Composite*) – Assemblage for which to calculate the fugacity.
- **reference_assemblage** (*burnman.Composite*) – Reference assemblage against which to measure the fugacity.

Returns

Value of the fugacity of the component in the assemblage with respect to the reference_assemblage.

Return type

float

burnman.tools.chemistry.**equilibrium_pressure**(*minerals*, *stoichiometry*, *temperature*, *pressure_initial_guess=100000.0*)

Given a list of minerals, their reaction stoichiometries and a temperature of interest, compute the equilibrium pressure of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **temperature** (*float*) – Temperature of interest [K].
- **pressure_initial_guess** (*float*) – Initial pressure guess [Pa].

Returns
The equilibrium pressure of the reaction [Pa].

Return type
float

burnman.tools.chemistry.**equilibrium_temperature**(*minerals*, *stoichiometry*, *pressure*, *temperature_initial_guess=1000.0*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.
- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**invariant_point**(*minerals_r1*, *stoichiometry_r1*, *minerals_r2*, *stoichiometry_r2*, *pressure_temperature_initial_guess=[1000000000.0, 1000.0]*)

Given a list of minerals, their reaction stoichiometries and a pressure of interest, compute the equilibrium temperature of the reaction.

Parameters

- **minerals** (list of *burnman.Mineral*) – List of minerals involved in the reaction.

- **stoichiometry** (*list of floats*) – Reaction stoichiometry for the minerals provided. Reactants and products should have the opposite signs [mol].
- **pressure** (*float*) – Pressure of interest [Pa].
- **temperature_initial_guess** (*float*) – Initial temperature guess [K].

Returns
The equilibrium temperature of the reaction [K].

Return type
float

burnman.tools.chemistry.**hugoniot**(*mineral*, *P_ref*, *T_ref*, *pressures*, *reference_mineral=None*)

Calculates the temperatures (and volumes) along a Hugoniot as a function of pressure according to the Hugoniot equation U2-U1 = 0.5*(p2 - p1)(V1 - V2) where U and V are the internal energies and volumes (mass or molar) and U = F + TS

Parameters

- **mineral** (*burnman.Mineral*) – Mineral for which the Hugoniot is to be calculated.
- **P_ref** (*float*) – Reference pressure [Pa]
- **T_ref** (*float*) – Reference temperature [K]
- **pressures** (*numpy.array of floats*) – Set of pressures [Pa] for which the Hugoniot temperature and volume should be calculated.
- **reference_mineral** (*burnman.Mineral*) – Mineral which is stable at the reference conditions Provides an alternative U_0 and V_0 when the reference mineral transforms to the mineral of interest at some (unspecified) pressure.

Returns
The Hugoniot temperatures and volumes at the given pressures.

Return type
tuple of numpy.arrays

burnman.tools.chemistry.**reactions_from_stoichiometric_matrix**(*stoichiometric_matrix*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters
stoichiometric_matrix (*2D numpy array*) – An array of the stoichiometric (molar) amounts of component j in compound i.

Returns
An array of the stoichiometric (molar) amounts of compound j in reaction i.

Return type
2D numpy array

burnman.tools.chemistry.**reactions_from_formulae**(*formulae*, *compound_names*, *return_strings=True*)

Returns a list of all the balanced reactions between compounds of fixed chemical composition. Includes both the forward and reverse reactions (so there will always be an even number of reactions).

Parameters

- **formulae** (*list of dictionaries or list of strings*) – List of the chemical formulae, either as strings or as a list of dictionaries of elements.
- **compound_names** (*list of strings*) – List of the compound names in the formula list.
- **return_strings** (*bool*) – Whether to return the reactions as strings or array.

Returns

Either a 2D array of the stoichiometric (molar) amounts of compound j in reaction i, or a list of strings. The parameter compound_names is only used if strings are requested.

Return type

2D numpy array or list of strings

20.23.7 Partitioning

burnman.tools.partitioning.**calculate_nakajima_fp_pv_partition_coefficient**(*pressure*, *temperature*, *bulk_composition_mol*, *initial_distribution_coefficien*

Calculate the partitioning of iron between periclase and bridgmanite as given by Nakajima et al., 2012.

Parameters

- **pressure** (*float*) – Equilibrium pressure [Pa]
- **temperature** (*float*) – Equilibrium temperature [K]
- **bulk_composition_mol** (*dict*) – Bulk composition [mol]. Only Mg, Fe, and Si are assumed to explicitly affect the partitioning with Al playing an implicit role.
- **initial_distribution_coefficient** (*float*) – The distribution coefficient (Kd_0) at 25 GPa and 0 K.

Returns

The proportion of Fe in ferropericlase and perovskite, respectively

Return type

tuple

20.23.8 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*, *n_rots=181*, *plot_vectors=True*, *ranges=None*)

Parameters

- **mineral** (*burnman.Mineral*) – Mineral object on which calculations should be done
- **plot_types** (*list of str*) – Plot types must be one of the following * 'vp' - V$_{P}$ (km/s) * 'vs1' - 'V$_{S1}$ (km/s) * 'vs2' - V$_{S2}$ (km/s) * 'vp/vs1' - V$_{P}$/V$_{S1}$ * 'vp/vs2' - V$_{P}$/V$_{S2}$ * 's anisotropy' - S-wave anisotropy (%), 200(vs1s - vs2s)/(vs1s + vs2s)) * 'linear compressibility' - Linear compressibility (GPa^{-1}) * 'youngs modulus' - Youngs Modulus (GPa) * 'minimum poisson ratio' - Minimum Poisson ratio * 'maximum poisson ratio' - Maximum Poisson ratio
- **axes** (*matplotlib.pyplot.axes objects*) – axes objects to be modified. Must be initialised with projection='polar'.
- **n_zenith** (*int*) – Number of zeniths (plot resolution).
- **n_azimuth** (*int*) – Number of azimuths (plot resolution).
- **n_divs** (*int*) – Number of divisions for the color scale.
- **n_rots** (*int*) – Number of along-vector rotations to find minimum and maximum Poisson ratios.
- **plot_vectors** (*bool*) – Whether or not to plot vectors for V_S and Poisson axial directions
- **ranges** (*list of (float, float) tuples, optional*) – Optional list of (min, max) tuples for each plot type. If not provided, the min and max of the data will be used.

Returns

Contour sets, ticks and line segments for each plot type.

Return type

(list of matplotlib.contour.QuadContourSet, list of list of float, list of numpy.ndarray)

20.23.9 Seismic

burnman.tools.seismic.**attenuation_correction**(*v_p*, *v_s*, *v_phi*, *Qs*, *Qphi*)

Applies the attenuation correction following [MBR+07], 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.tools.seismic.attenuation_correction()*

Parameters

- **v_p** (*float*) – P wave velocity in [m/s].
- **v_s** (*float*) – S wave velocitiy in [m/s].
- **v_phi** (*float*) – Bulk sound velocity in [m/s].
- **Qs** (*float*) – Shear quality factor [dimensionless].
- **Qphi** (*float*) – Bulk quality factor [dimensionless].

Returns
Corrected P wave, S wave and bulk sound velocities in [m/s].

Return type
tuple

20.23.10 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. Helper function which uses tvel_formatted_data_and_header()

Parameters

- **planet_or_layer** (*burnman.Planet* or *burnman.Layer*.) – Planet or layer to write out to tvel file
- **modelname** (*str*) – Basename of file to write to (suffix .tvel added by function).
- **background_model** (burnman.seismic.Seismic1DModel) – 1D seismic model to fill in parts of planet (likely to be an earth model) that aren't defined by layer (only need when using *burnman.Layer*)

burnman.tools.output_seismo.**write_axisem_input**(*layers*, *modelname='burnmanmodel_foraxisem'*, *axisem_ref='axisem_prem_ani_noocean.txt'*, *plotting=False*, *verbose=True*)

Function to write input file for AXISEM (www.axisem.info). Helper function which uses axisem_formatted_data_and_reference()

The input can be a single layer, or a list of layers taken from a planet (planet.layers). Currently this function will implement explicit discontinuities between layers in the seismic model. Currently this function is only set for Earth.

Parameters

- **layers** (list of one or more *burnman.Layer*) – List of layers to put in AXISEM file.
- **modelname** (*str*) – Name of model, appears in name of output file.
- **axisem_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

burnman.tools.output_seismo.**write_mineos_input**(*layers*, *modelname='burnmanmodel_formineos'*, *mineos_ref='mineos_prem_noocean.txt'*, *plotting=False*, *verbose=True*)

Function to write input file for Mineos (https://geodynamics.org/cig/software/mineos/). Helper function which uses mineos_formatted_data_and_reference()

Note: currently, this function only honors the discontinuities already in the synthetic input file, so it is best to only replace certain layers with burnman values

Parameters

- **layers** (list of one or more *burnman.Layer*) – List of layers to put in Mineos file.
- **modelname** (*str*) – Name of model, appears in name of output file.
- **mineos_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

burnman.tools.output_seismo.**tvel_formatted_data_and_header**(*planet_or_layer*, *background_model=None*)

Formats data and creates a header 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
- **background_model** (burnman.seismic.Seismic1DModel) – 1D seismic model to fill in parts of planet (likely to be an earth model) that aren't defined by layer (only need when using *burnman.Layer*)

burnman.tools.output_seismo.**axisem_formatted_data_and_reference**(*layers*, *axisem_ref='axisem_prem_ani_noocean'*, *plotting=False*)

Formats data for AXISEM (www.axisem.info). 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.
- **axisem_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

burnman.tools.output_seismo.**mineos_formatted_data_and_reference**(*layers*, *mineos_ref='mineos_prem_noocean.txt'*, *plotting=False*)

Formats data and reference data for Mineos (https://geodynamics.org/cig/software/mineos/).

Note: currently, this function only honors the discontinuities already in the synthetic input file, so it is best to only replace certain layers with burnman values

Parameters

- **layers** (list of one or more *burnman.Layer*) – List of layers to put in Mineos file.
- **mineos_ref** (*str*) – Reference file, used to copy the header and for the rest of the planet, in the case of a *burnman.Layer*.
- **plotting** (*bool*) – Choose whether to show plot of the old model and replaced model.

20.23.11 Polytopes

burnman.tools.polytope.**solution_polytope_from_charge_balance**(*charges*, *charge_total*, *return_fractions=False*)

Creates a polytope object from a list of the charges for each species on each site and the total charge for all site-species.

Parameters

- **charges** (*2D list of floats*) – 2D list containing the total charge for species j on site i, including the site multiplicity. So, for example, a solution with the site formula [Mg,Fe]3[Mg,Al,Si]2Si3O12 would have the following list: [[6., 6.], [4., 6., 8.]].
- **charge_total** (*float*) – The total charge for all site-species per formula unit. The example given above would have charge_total = 12.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats. Default is False.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**solution_polytope_from_endmember_occupancies**(*endmember_occupancies*, *return_fractions=False*)

Creates a polytope object from a list of independent endmember occupancies.

Parameters

- **endmember_occupancies** (*2D numpy array*) – 2D list containing the site-species occupancies j for endmember i. So, for example, a solution with independent endmembers [Mg]3[Al]2Si3O12, [Mg]3[Mg0.5Si0.5]2Si3O12, [Fe]3[Al]2Si3O12 might have the following array: [[1., 0., 1., 0., 0.], [1., 0., 0., 0.5, 0.5], [0., 1., 1., 0., 0.]], where the order of site-species is Mg_A, Fe_A, Al_B, Mg_B, Si_B.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**composite_polytope_at_constrained_composition**(*composite*, *composition*, *return_fractions=False*)

Creates a polytope object from a Composite object and a composition. This polytope describes the complete set of valid composite endmember amounts that satisfy the compositional constraints.

Parameters

- **composite** (*burnman.Composite* object) – A composite containing one or more Solution and Mineral objects.
- **composition** (*dict*) – A dictionary containing the amounts of each element.
- **return_fractions** (*bool*) – Determines whether the created polytope object returns its attributes (such as endmember occupancies) as fractions or as floats.

Returns
A polytope object corresponding to the parameters provided.

Return type
burnman.polytope.MaterialPolytope object

burnman.tools.polytope.**reorder_to_echelon**(*A*)

Reorders the rows of a matrix A so that it is in echelon form. :param A: The input matrix. :type A: 2D numpy array :returns: The reordered matrix. :rtype: 2D numpy array

burnman.tools.polytope.**simplify_composite_with_composition**(*composite*, *composition*)

Takes a composite and a composition, and returns the simplest composite object that spans the solution space at the given composition.

For example, if the composition is given as {'Mg': 2., 'Si': 1.5, 'O': 5.}, and the composite is given as a mix of Mg,Fe olivine and pyroxene solutions, this function will return a composite that only contains the Mg-bearing endmembers.

If the solutions have endmember proportions that are consistent with the bulk composition, the site occupancies of the new solution models are set to the values in the original models.

Parameters

- **composite** (*burnman.Composite* object) – The initial Composite object.
- **composition** (*dict*) – A dictionary containing the amounts of each element.

Returns

The simplified Composite object

Return type

burnman.Composite object

burnman.tools.polytope.**greedy_independent_endmember_selection**(*endmember_site_occupancies*, *site_occupancies*, *small_fraction_tol=0.0*, *norm_tol=1e-12*)

Greedy algorithm to select independent endmembers from a solution to approximate given site occupancies through a non-negative linear combination of endmember site occupancies.

This function starts with the full site occupancies and then iteratively selects endmembers to approximate those site occupancies. It loops through all possible endmembers in a number of steps. At each step the algorithm selects the endmember that can be subtracted in the largest amount from the current residual site occupancies without making any site occupancy negative. The process continues until either no endmember can be subtracted in an amount greater than fraction_tol, or the norm of the residual site occupancies is less than tol.

Parameters

- **endmember_site_occupancies** (*np.ndarray*) – A 2D array of shape (m, n), where m is the number of endmembers and n is the number of sites. Each row corresponds to the site occupancies of an endmember.
- **site_occupancies** (*np.ndarray*) – A 1D array of length n, representing the target site occupancies to approximate.
- **small_fraction_tol** (*float, optional, default 0.0*) – Algorithm stops if no endmember can be added with a molar fraction larger than this value.
- **norm_tol** (*float, optional, default 1e-12*) – Algorithm stops if the norm of the residual site occupancies is less than this value.

Returns

indices of selected endmembers, their fractions, and the final residual site occupancies.

Return type
tuple(list[int], list[float], np.ndarray)

20.24 Utilities

BurnMan has a number of low-level utilities to help achieve common goals.

20.24.1 Unit cell

burnman.utils.unitcell.**molar_volume_from_unit_cell_volume**(*unit_cell_v*, *z*)

Converts a unit cell volume from Angstroms^3 per unitcell, to m^3/mol.

Parameters

- **unit_cell_v** (*float*) – Unit cell volumes [A^3/unit cell].
- **z** (*float*) – Number of formula units per unit cell.

Returns
Volume [m^3/mol]

Return type
float

burnman.utils.unitcell.**cell_parameters_to_vectors**(*cell_parameters*, *frame_convention*)

Converts cell parameters to unit cell vectors.

Parameters

- **cell_parameters** (*numpy.array (1D)*) – An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.
- **frame_convention** (*list of three integers*) – A list (c) defining the reference frame convention. This function dictates that the c[0]th cell vector is colinear with the c[0]th axis, the c[1]th cell vector is perpendicular to the c[2]th axis, and the c[2]th cell vector is defined to give a right-handed coordinate system. In common crystallographic shorthand, x[c[0]] // a[c[0]], x[c[2]] // a[c[2]]^* (i.e. perpendicular to a[c[0]] and a[c[1]]).

Returns
The three vectors defining the parallelopiped cell [m].

Return type
numpy.array (2D)

burnman.utils.unitcell.**rotation_to_crystallographic_frame**(*cell_vectors*, *frame_convention*)

Returns a rotation matrix Q that rotates a set of cell vectors to the crystallographic frame defined by a given frame convention via the transformation: cell_vectors (xtl) = (Q @ cell_vectors.T).T

Parameters

- **cell_vectors** (*numpy.array (2D)*) – The three vectors defining the parallelopiped cell [m].
- **frame_convention** (*list of three integers*) – A list (c) defining the reference frame convention. This function dictates that the c[0]th cell vector is colinear with the c[0]th axis, the c[1]th cell vector is perpendicular to the c[2]th axis, and the c[2]th cell vector is defined to give a right-handed coordinate system. In common crystallographic shorthand, x[c[0]] // a[c[0]], x[c[2]] // a[c[2]]^* (i.e. perpendicular to a[c[0]] and a[c[1]]).

Returns
The requested rotation matrix.

Return type
numpy.array (2D)

burnman.utils.unitcell.**cell_vectors_to_parameters**(*vectors*, *frame_convention*, *rotate=False*)

Converts unit cell vectors to cell parameters.

Parameters

- **vectors** (*numpy.array (2D)*) – The three vectors defining the parallelopiped cell [m].
- **frame_convention** (*list of three integers*) – A list (c) defining the reference frame convention. This function dictates that the c[0]th cell vector is colinear with the c[0]th axis, the c[1]th cell vector is perpendicular to the c[2]th axis, and the c[2]th cell vector is defined to give a right-handed coordinate system. In common crystallographic shorthand, x[c[0]] // a[c[0]], x[c[2]] // a[c[2]]^* (i.e. perpendicular to a[c[0]] and a[c[1]]).
- **rotate** (*bool*) – If True, the input vectors are rotated to satisfy the requested frame convention. If False, the input vectors are assumed to already satisfy the requested frame convention.

Returns
An array containing the three lengths of the unit cell vectors [m], and the three angles [degrees]. The first angle (α) corresponds to the angle between the second and the third cell vectors, the second (β) to the angle between the first and third cell vectors, and the third (γ) to the angle between the first and second vectors.

Return type
numpy.array (1D)

20.24.2 Chemistry

burnman.utils.chemistry.**IUPAC_element_order** = ['v', 'Og', 'Rn', 'Xe', 'Kr', 'Ar', 'Ne', 'He', 'Fr', 'Cs', 'Rb', 'K', 'Na', 'Li', 'Ra', 'Ba', 'Sr', 'Ca', 'Mg', 'Be', 'Lr', 'No', 'Md', 'Fm', 'Es', 'Cf', 'Bk', 'Cm', 'Am', 'Pu', 'Np', 'U', 'Pa', 'Th', 'Ac', 'Lu', 'Yb', 'Tm', 'Er', 'Ho', 'Dy', 'Tb', 'Gd', 'Eu', 'Sm', 'Pm', 'Nd', 'Pr', 'Ce', 'La', 'Y', 'Sc', 'Rf', 'Hf', 'Zr', 'Ti', 'Db', 'Ta', 'Nb', 'V', 'Sg', 'W', 'Mo', 'Cr', 'Bh', 'Re', 'Tc', 'Mn', 'Hs', 'Os', 'Ru', 'Fe', 'Mt', 'Ir', 'Rh', 'Co', 'Ds', 'Pt', 'Pd', 'Ni', 'Rg', 'Au', 'Ag', 'Cu', 'Cn', 'Hg', 'Cd', 'Zn', 'Nh', 'Tl', 'In', 'Ga', 'Al', 'B', 'Fl', 'Pb', 'Sn', 'Ge', 'Si', 'C', 'Mc', 'Bi', 'Sb', 'As', 'P', 'N', 'Lv', 'Po', 'Te', 'Se', 'S', 'O', 'H', 'Ts', 'At', 'I', 'Br', 'Cl', 'F']

IUPAC_element_order is a list of all the elements. Element order is based loosely on electronegativity, following the scheme suggested by IUPAC, except that H comes after the Group 16 elements, not before them.

burnman.utils.chemistry.**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}

atomic_masses is a dictionary of atomic masses

burnman.utils.chemistry.**dictionarize_formula**(*formula*)

A function to read a chemical formula string and convert it into a dictionary

Parameters

formula (*str*) – Chemical formula, written in the XnYm format, where the formula has n atoms of element X and m atoms of element Y

Returns

The same chemical formula, but expressed as a dictionary.

Return type

dict

burnman.utils.chemistry.**sum_formulae**(*formulae*, *amounts=None*)

Adds together a set of formulae.

Parameters

- **formulae** (*list of dictionary or counter objects*) – List of chemical formulae.
- **amounts** (*list of floats*) – List of amounts of each formula.

Returns
The sum of the user-provided formulae

Return type
Counter object

burnman.utils.chemistry.**formula_mass**(*formula*)

A function to take a chemical formula and compute the formula mass.

Parameters
formula (*dict or Counter object*) – A chemical formula

Returns
The mass per mole of formula [kg]

Return type
float

burnman.utils.chemistry.**convert_formula**(*formula*, *to_type='mass'*, *normalize=False*)

Converts a chemical formula from one type (mass or molar) into the other. Renormalises amounts if normalize=True.

Parameters

- **formula** (*dict or Counter object*) – A chemical formula.
- **to_type** (*str*) – Conversion type, one of 'mass' or 'molar'.
- **normalize** (*bool*) – Whether or not to normalize the converted formula to 1.

Returns
The converted formula.

Return type
dict

burnman.utils.chemistry.**process_solution_chemistry**(*solution_model*)

This function parses a class instance with a "site_formulae" attribute containing site information, e.g.

['[Mg]3[Al]2Si3O12', '[Mg]3[Mg1/2Si1/2]2Si3O12']

It outputs the bulk composition of each endmember (removing the site information), and also a set of variables and arrays which contain the site information. These are output in a format that can easily be used to calculate activities and gibbs free energies, given molar fractions of the phases and pressure and temperature where necessary.

Parameters
solution_model – Class must have a "site_formulae" attribute, containing a list of chemical formulae with site information

Return type
None

Note

Nothing is returned from this function, but the solution_model object gains the following attributes:

- **empty_formula [string]**
  Abbreviated chemical formula with sites denoted by empty square brackets.
- **general_formula [string]**
  General chemical formula with sites denoted by square brackets filled with a comma-separated list of species
- **n_sites [integer]**
  Number of sites in the solution. Should be the same for all endmembers.
- **sites [list of lists of strings]**
  A list of species for each site in the solution.
- **site_names [list of strings]**
  A list of species_site pairs in the solution, where each distinct site is given by a unique uppercase letter e.g. ['Mg_A', 'Fe_A', 'Al_A', 'Al_B', 'Si_B'].
- **n_occupancies [integer]**
  Sum of the number of possible species on each of the sites in the solution. Example: A binary solution [[A][B],[B][C1/2D1/2]] would have n_occupancies = 5, with two possible species on Site 1 and three on Site 2.
- **site_multiplicities [2D array of floats]**
  A 1D array for each endmember in the solution, containing the multiplicities of each site per formula unit. To simplify computations later, the multiplicities are repeated for each species on each site, so the shape of this attribute is (n_endmembers, n_site_species).
- **endmember_occupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the fraction of atoms of each species on each site.
- **endmember_noccupancies [2d array of floats]**
  A 1D array for each endmember in the solution, containing the number of atoms of each species on each site per mole of endmember.

burnman.utils.chemistry.**site_occupancies_to_strings**(*site_species_names*, *site_multiplicities*, *endmember_occupancies*)

Converts a list of endmember site occupancies into a list of string representations of those occupancies.

Parameters

- **site_species_names** (*2D list of strings*) – A list of list of strings, giving the names of the species which reside on each site. List of sites, each of which contains a list of the species occupying each site.

- **site_multiplicities** (*1D or 2D numpy array of floats*) – List of floats giving the multiplicity of each site. If 2D, must have the same shape as endmember_occupancies. If 1D, must be either the same length as the number of sites, or the same length as site_species_names (with an implied repetition of the same number for each species on a given site).
- **endmember_occupancies** (*2D numpy array of floats*) – A list of site-species occupancies for each endmember. The first dimension loops over the endmembers, and the second dimension loops over the site-species occupancies for that endmember. The total number and order of occupancies must be the same as the strings in site_species_names.

Returns
A list of strings in standard burnman format. For example, [Mg]3[Al]2 would correspond to the classic two-site pyrope garnet.

Return type
list of strings

burnman.utils.chemistry.**compositional_array**(*formulae*)

Parameters
formulae (*list of dicts*) – List of chemical formulae

Returns
Array of endmember formulae and a list of elements.

Return type
2D numpy.array of floats and a list of strs

burnman.utils.chemistry.**ordered_compositional_array**(*formulae*, *elements*)

Parameters
formulae (*list of dicts*) – List of chemical formulae

:param elements : List of elements :type elements: list of strings

Returns
Array of endmember formulae

Return type
2D array of floats

burnman.utils.chemistry.**formula_to_string**(*formula*, *use_fractions=True*, *significant_figures=2*, *latex=False*)

Parameters

- **formula** (*dict or Counter*) – Chemical formula
- **use_fractions** (*bool*) – Whether or not to use fractions in the output string. If False, numbers are rounded to the number of significant figures given by the significant_figures parameter.

- **significant_figures** (*int*) – Number of significant figures to use if use_fractions=False.
- **latex** (*bool*) – Whether or not to format the output string for use in LaTeX.

Returns
A formula string, with element order as given in the list IUPAC_element_order. If one or more keys in the dictionary are not one of the elements in the periodic table, then they are added at the end of the string.

Return type
str

burnman.utils.chemistry.**sort_element_list_to_IUPAC_order**(*element_list*)

:param element_list : List of elements. :type element_list: list

Returns
List of elements sorted into IUPAC order

Return type
list

burnman.utils.chemistry.**convert_fractions**(*composite*, *phase_fractions*, *input_type*, *output_type*)

Takes a composite with a set of user defined molar, volume or mass fractions (which do not have to be the fractions currently associated with the composite) and converts the fractions to molar, mass or volume.

Conversions to and from mass require a molar mass to be defined for all phases. Conversions to and from volume require set_state to have been called for the composite.

Parameters

- **composite** (*Composite*) – Composite for which fractions are to be defined.
- **phase_fractions** (*list of floats*) – List of input phase fractions (of type input_type).
- **input_type** (*str*) – Input fraction type. One of 'molar', 'mass' or 'volume'.
- **output_type** (*str*) – Output fraction type. One of 'molar', 'mass' or 'volume'.

Returns
List of output phase fractions (of type output_type)

Return type
list of floats

burnman.utils.chemistry.**reaction_matrix_as_strings**(*reaction_matrix*, *compound_names*)

Returns a list of string representations of all the reactions in reaction_matrix.

Parameters

- **reaction_matrix** (*2D numpy array*) – Matrix of stoichiometric amounts of each compound j in reaction i.

- **compound_names** (*list of strings*) – List of compound names.

Returns
List of strings corresponding to each reaction.

Return type
list of strings

20.24.3 Anisotropy

burnman.utils.anisotropy.**voigt_index_to_ij**(*m*)

Returns the ij (or kl) indices of the stiffness tensor which correspond to those of the Voigt notation m (or n).

burnman.utils.anisotropy.**voigt_notation_to_stiffness_tensor**(*voigt_notation*)

Converts a stiffness tensor in Voigt notation (6x6 matrix) to the full fourth rank tensor (3x3x3x3 matrix).

burnman.utils.anisotropy.**voigt_notation_to_compliance_tensor**(*voigt_notation*)

burnman.utils.anisotropy.**contract_stresses**(*stresses*)

Takes a stress tensor in standard (3x3) form and returns the Voigt form (6). No factors are required to maintain the relationship with the corresponding stiffness and strain tensors (see contract_strains, which does require multiplicative factors).

burnman.utils.anisotropy.**expand_stresses**(*stresses*)

Takes a stress tensor in Voigt form (6) and returns the standard form (3x3). No factors are required to maintain the relationship with the corresponding stiffness and strain tensors (see contract_strains, which does require multiplicative factors).

burnman.utils.anisotropy.**contract_strains**(*strains*)

Takes a stress tensor in standard (3x3) form and returns the Voigt form (6).

The off-diagonal elements are multiplied by 2 in the conversion from standard form to Voigt form to maintain the correct relationship with the corresponding stiffness and compliance tensors, which are defined in terms of the engineering strain.

burnman.utils.anisotropy.**expand_strains**(*strains*)

Takes a stress tensor in Voigt form (6) and returns the standard form (3x3).

The last three elements are divided by 2 in the conversion from Voigt form to standard form to maintain the correct relationship with the corresponding stiffness and compliance tensors.

burnman.utils.anisotropy.**contract_compliances**(*compliances*)

Takes a compliance tensor in standard (3x3x3x3) form and returns the Voigt form (6x6). Note the compliance factors which are required to maintain the inverse relationship with the corresponding stiffness tensor.

burnman.utils.anisotropy.**contract_stiffnesses**(*stiffnesses*)

Takes a stiffness tensor in standard (3x3x3x3) form and returns the Voigt form (6x6).

burnman.utils.anisotropy.**voigt_array_from_cijs**(*cijs*, *index_lists*)

Takes a list of cijs and a list of list of tuples corresponding to the positions of each cij in the Voigt form matrix. Note that the indices run from 0–5, not 1–6.

burnman.utils.anisotropy.**polar_decomposition**(*F*, *strain_measure='U'*)

Perform polar decomposition of a deformation gradient F into a rotation matrix R and a strain measure: U (right stretch), V (left stretch), or H (right Hencky strain).

Parameters

- **F** (*np.ndarray*) – Deformation gradient (3x3)
- **strain_measure** (*str*) – The type of strain measure to return.

Returns

Rotation matrix R and the specified strain measure.

Return type

tuple of (np.ndarray, np.ndarray)

20.24.4 Plotting

class burnman.utils.plotting.**Cell**(*x*, *y*, *h*, *polygon*)

Bases: object

Represents a square cell used during the polygon center search.

Parameters

- **x** (*float*) – X-coordinate of the cell center.
- **y** (*float*) – Y-coordinate of the cell center.
- **h** (*float*) – Half-size of the cell.
- **polygon** (*list of numpy arrays*) – The input polygon as a list of rings.

burnman.utils.plotting.**closest_point_on_segment**(*p*, *a*, *b*)

Return the closest point on segment ab to point p.

Parameters

- **p** (*np.array of shape (2,)*) – the target point
- **a** (*np.array of shape (2,)*) – the start point of the segment
- **b** (*np.array of shape (2,)*) – the end point of the segment

Returns

closest point on segment ab

Return type

np.array of shape (2,)

burnman.utils.plotting.**closest_point_on_polygon**(*p*, *polygon*)

Find the closest point on a polygon to point p.

Parameters

- **p** (*np.array of shape (2,)*) – the target point
- **polygon** (*np.array of shape (N, 2)*) – the polygon vertices (assumed closed or will be treated as closed)

Returns
closest point on polygon

Return type
np.array of shape (2,)

burnman.utils.plotting.**visual_center_of_polygon**(*polygon_rings*, *precision=1.0*, *with_distance=False*)

Compute the pole of inaccessibility (visual center) of a polygon with the specified precision.

Parameters

- **polygon_rings** (*list of numpy arrays of shape (N, 2)*) – A polygon represented as a list of rings.
- **precision** (*float*) – Desired precision. Stops when improvement is less than this value.
- **with_distance** (*bool*) – If True, also return the distance to the closest edge.

Returns
The [x, y] coordinates of the center, and optionally the distance.

Return type
list or tuple

20.24.5 Mathematical

burnman.utils.math.**round_to_n**(*x*, *xerr*, *n*)

burnman.utils.math.**unit_normalize**(*a*, *order=2*, *axis=-1*)

Calculates the L2 normalized array of numpy array a of a given order and along a given axis.

burnman.utils.math.**linear_interpol**(*x*, *x1*, *x2*, *y1*, *y2*)

Linearly interpolate to point x, between the points (x1,y1), (x2,y2)

burnman.utils.math.**bracket**(*fn*, *x0*, *dx*, *args=()*, *ratio=1.618*, *maxiter=100*)

Given a function and a starting guess, find two inputs for the function that bracket a root.

Parameters

- **fn** (*function*) – The function to bracket.
- **x0** (*float*) – The starting guess.
- **dx** (*float*) – Small step for starting the search.
- **args** (*tuple*) – Additional arguments to give to fn.

- **ratio** (*float*) – The step size increases by this ratio every step in the search. Defaults to the golden ratio.
- **maxiter** (*int*) – The maximum number of steps before giving up.

Returns
xa, xb, fa, fb. xa and xb are the inputs which bracket a root of fn. fa and fb are the values of the function at those points. If the bracket function takes more than maxiter steps, it raises a ValueError.

Return type
tuple of floats

burnman.utils.math.**smooth_array**(*array*, *grid_spacing*, *gaussian_rms_widths*, *truncate=4.0*, *mode='inverse_mirror'*)

Creates a smoothed array by convolving it with a gaussian filter. Grid resolutions and gaussian RMS widths are required for each of the axes of the numpy array. The smoothing is truncated at a user-defined number of standard deviations. The edges of the array can be padded in a number of different ways given by the 'mode' parameter.

Parameters

- **array** (*numpy.ndarray*) – The array to smooth.
- **grid_spacing** (*numpy.array of floats*) – The spacing of points along each axis.
- **gaussian_rms_widths** (*numpy.array of floats*) – The Gaussian RMS widths/standard deviations for the Gaussian convolution.
- **truncate** (*float*) – The number of standard deviations at which to truncate the smoothing.
- **mode** (*str*) – {'reflect', 'constant', 'nearest', 'mirror', 'wrap', 'inverse_mirror'} The mode parameter determines how the array borders are handled either by scipy.ndimage.filters.gaussian_filter. Default is 'inverse_mirror', which uses `burnman.tools.math._pad_ndarray_inverse_mirror()`.

Returns
The smoothed array

Return type
numpy.ndarray

burnman.utils.math.**interp_smoothed_array_and_derivatives**(*array*, *x_values*, *y_values*, *x_stdev=0.0*, *y_stdev=0.0*, *truncate=4.0*, *mode='inverse_mirror'*, *indexing='xy'*)

Creates a smoothed array on a regular 2D grid. Smoothing is achieved using `burnman.tools.math.smooth_array()`. Outputs scipy.interpolate.RegularGridInterpolator() interpolators which can be used to query the array, or its derivatives in the x- and y- directions.

Parameters

- **array** (*numpy.array (2D)*) – The array to smooth. Each element array[i][j] corresponds to the position x_values[i], y_values[j]
- **x_values** (*numpy.array (1D)*) – The gridded x values over which to create the smoothed grid
- **y_values** (*numpy.array (1D)*) – The gridded y_values over which to create the smoothed grid
- **x_stdev** (*float*) – The standard deviation for the Gaussian filter along the x axis
- **y_stdev** (*float*) – The standard deviation for the Gaussian filter along the y axis
- **truncate** (*float*) – The number of standard deviations at which to truncate the smoothing.
- **mode** (*str*) – {'reflect', 'constant', 'nearest', 'mirror', 'wrap', 'inverse_mirror'} The mode parameter determines how the array borders are handled either by scipy.ndimage.filters.gaussian_filter. Default is 'inverse_mirror', which uses `burnman.tools.math._pad_ndarray_inverse_mirror()`.
- **indexing** (*str*) – {'xy', 'ij'}, optional Cartesian ('xy', default) or matrix ('ij') indexing of output. See numpy.meshgrid for more details.

Returns
Three RegularGridInterpolator functors interpolation functions for the smoothed property and the first derivatives with respect to x and y.

Return type
tuple

`burnman.utils.math.`**l2_norm_profiles**(*depth*, *calc*, *obs*)

Computes the L2 norm for N profiles at a time (assumed to be linear between points).

Parameters

- **depths** (*array of float*) – depths. $[m]$
- **calc** (*list of arrays of float*) – N arrays calculated values, e.g. [mat_vs,mat_vphi]
- **obs** (*list of arrays of float*) – N arrays of values (observed or calculated) to compare to , e.g. [seis_vs, seis_vphi]

Returns
array of L2 norms of length N

Return type
array of floats

`burnman.utils.math.`**chisqr_profiles**(*calc*, *obs*)

Computes the chisqr factor for N profiles at a time. Assumes a 1% a priori uncertainty on the seismic model.

Parameters

- **calc** (*list of arrays of float*) – N arrays calculated values, e.g. [mat_vs,mat_vphi]
- **obs** (*list of arrays of float*) – N arrays of values (observed or calculated) to compare to, e.g. [seis_vs, seis_vphi]

Returns
error array of length N

Return type
array of floats

burnman.utils.math.**l2_norm_profile**(*x*, *funca*, *funcb*)

Computes the L2 norm of the difference between two 1D profiles, assuming linear interpolation between points.

This is equivalent to the square root of the integrated squared difference between the two functions over the given depth range. The integration is performed using the trapezoidal rule.

Parameters

- **x** (*array-like of float*) – Array of depth values [m], must be 1D and monotonically increasing.
- **funca** (*array-like of float*) – First profile (e.g., model or predicted values), must be same length as x.
- **funcb** (*array-like of float*) – Second profile (e.g., observed or reference values), same shape as funca.

Returns
L2 norm (scalar)

Return type
float

burnman.utils.math.**chisqr_profile**(*calc*, *obs*, *uncertainties=0.01*)

Computes the χ^2 factor for a single profile, assuming a 1 % uncertainty on the reference (observed) values. This provides a normalized measure of the deviation between calculated and reference values.

The χ^2 factor is calculated as:

$$\chi^2 = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\text{calc}_i - \text{obs}_i}{0.01 \cdot \text{mean}(\text{obs})} \right)^2$$

where N is the number of elements in the profile.

Parameters

- **calc** (*array*) – Array of calculated values (e.g., from a model).
- **obs** (*array*) – Array of reference or observed values.
- **uncertainties** (*array or float*) – Array of uncertainties values.

Returns
The χ^2 factor for the profile.

Return type
float

burnman.utils.math.**independent_row_indices**(*array*)

Returns the indices corresponding to an independent set of rows for a given array. The independent rows are determined from the pivots used during row reduction/Gaussian elimination.

Parameters
array (*2D numpy.array of floats*) – The input array.

Returns
The indices corresponding to a set of independent rows of the input array.

Return type
1D numpy array of integers

burnman.utils.math.**array_to_rational_matrix**(*array*)

Converts a numpy array into a sympy matrix filled with rationals

burnman.utils.math.**complete_basis**(*basis*, *flip_columns=True*)

Extends a partial basis (with fewer rows than columns) to a full basis by adding rows from the identity matrix that are linearly independent of the existing rows.

This is done by computing the reduced row echelon form (RREF) of the basis to identify pivot columns. Identity rows corresponding to non-pivot columns are added to complete the basis.

Parameters

- **basis** (*numpy.ndarray*) – A 2D NumPy array representing a partial basis, with shape (n, m) where n < m.
- **flip_columns** (*bool, optional*) – If True, the basis columns are flipped (reversed) before computing the RREF. This is helpful if the dependent columns are expected to appear later in the matrix. Default is True.

Returns
A completed basis as a NumPy array with shape (m, m), containing the original rows plus additional rows from the identity matrix.

Return type
numpy.ndarray

burnman.utils.math.**generate_complete_basis**(*incomplete_basis*, *array*)

Given a 2D array with independent rows and a second 2D array that spans a larger space, creates a complete basis for the combined array using all the rows of the first array, followed by any required rows of the second array. So, for example, if the first array is: [[1, 0, 0], [1, 1, 0]] and the second array is: [[1, 0, 0], [0, 1, 0], [0, 0, 1]], the complete basis will be: [[1, 0, 0], [1, 1, 0], [0, 0, 1]].

Parameters

- **incomplete_basis** (*2D numpy.array*) – An array containing the basis to be completed.
- **array** (*2D numpy.array*) – An array spanning the full space for which a basis is required.

Returns
An array containing the basis vectors spanning both of the input arrays.

Return type
2D numpy array

burnman.utils.math.**is_positive_definite**(*matrix*)

Checks if a matrix is positive definite

Parameters
matrix (*2D numpy array*) – Input matrix

Returns
Whether or not the matrix is positive definite

Return type
bool

burnman.utils.math.**generalised_gammainc**(*a*, *x1*, *x2*)

An implementation of the generalised incomplete gamma function. Computed using the relationship with the regularised lower incomplete gamma function (scipy.special.gammainc). Uses the recurrence relation wherever a<0.

We could have used mpmath.gammainc(a, x1, x2) directly, but it is significantly slower than this implementation.

20.24.6 Row reductions and independent rows

burnman.utils.reductions.**row_reduce**(*M*, *iszerofunc=<function <lambda>>*, *simpfunc=<function <lambda>>*, *normalize_last=True*, *normalize=True*, *zero_above=True*)

burnman.utils.reductions.**independent_row_indices**(*m*, *iszerofunc=<function <lambda>>*, *simpfunc=<function <lambda>>*)

20.24.7 Miscellaneous

class burnman.utils.misc.**OrderedCounter**(*iterable=None*, */ (Positional-only parameter separator (PEP 570))*, ***kwds*)

Bases: Counter, OrderedDict

Counter that remembers the order elements are first encountered

clear(*/*)

Remove all items from ordered dict.

copy()

Return a shallow copy.

elements()

Iterator over elements repeating each as many times as its count.

```
>>> c = Counter('ABCABC')
>>> sorted(c.elements())
['A', 'A', 'B', 'B', 'C', 'C']
```

Knuth's example for prime factors of 1836: 2**2 * 3**3 * 17**1

```
>>> import math
>>> prime_factors = Counter({2: 2, 3: 3, 17: 1})
>>> math.prod(prime_factors.elements())
1836
```

Note, if an element's count has been set to zero or is a negative number, elements() will ignore it.

classmethod fromkeys(*iterable*, *v=None*)

Create a new ordered dictionary with keys from iterable and values set to value.

get(*key*, *default=None*, */*)

Return the value for key if key is in the dictionary, else default.

items(*/*)

Return a set-like object providing a view on the dict's items.

keys(*/*)

Return a set-like object providing a view on the dict's keys.

most_common(*n=None*)

List the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
>>> Counter('abracadabra').most_common(3)
[('a', 5), ('b', 2), ('r', 2)]
```

move_to_end(*/*, *key*, *last=True*)

Move an existing element to the end (or beginning if last is false).

Raise KeyError if the element does not exist.

pop(*/*, *key*, *default=<unrepresentable>*)

If the key is not found, return the default if given; otherwise, raise a KeyError.

popitem(*/*, *last=True*)

Remove and return a (key, value) pair from the dictionary.

Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*/*, *key*, *default=None*)

Insert key with a value of default if key is not in the dictionary.

Return the value for key if key is in the dictionary, else default.

subtract(*iterable=None*, */*, ***kwds*)

Like dict.update() but subtracts counts instead of replacing them. Counts can be reduced below zero. Both the inputs and outputs are allowed to contain zero and negative counts.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.subtract('witch')             # subtract elements from another␣
↪iterable
>>> c.subtract(Counter('watch'))    # subtract elements from another␣
↪counter
>>> c['h']                          # 2 in which, minus 1 in witch,␣
↪minus 1 in watch
0
>>> c['w']                          # 1 in which, minus 1 in witch,␣
↪minus 1 in watch
-1
```

total()

Sum of the counts

update(*iterable=None*, */*, ***kwds*)

Like dict.update() but add counts instead of replacing them.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.update('witch')           # add elements from another iterable
>>> d = Counter('watch')
>>> c.update(d)                 # add elements from another counter
>>> c['h']                      # four 'h' in which, witch, and watch
4
```

values(*/*)

Return an object providing a view on the dict's values.

burnman.utils.misc.**copy_documentation**(*copy_from*)

Decorator @copy_documentation(another_function) will copy the documentation found in a different function (for example from a base class). The docstring applied to some function a() will be

```
(copied from BaseClass.some_function):
```

<documentation from BaseClass.some_function> <optionally the documentation found in a()>

burnman.utils.misc.**merge_two_dicts**(*x*, *y*)

Given two dicts, merge them into a new dict as a shallow copy.

burnman.utils.misc.**flatten**(*arr*)

burnman.utils.misc.**pretty_string_values**(*popt*, *pcov*, *extra_decimal_places=0*, *combine_value_and_sigma=False*)

Takes a numpy array of parameters, the corresponding covariance matrix and a set of parameter names and returns the scaled variables and principal 1-s.d.uncertainties (np.sqrt(pcov[i][i])) and scaling factor as three separate lists of strings.

Parameters

- **popt** (*numpy array*) – Parameter values
- **pcov** (*2D numpy array*) – Variance-covariance matrix
- **extra_decimal_places** (*int, optional*) – extra precision for values, defaults to 0
- **combine_value_and_sigma** (*bool, optional*) – give values in value(sigma) format, defaults to False

Returns

values, uncertainties and the scaling factors

Return type

tuple of 3 lists

burnman.utils.misc.**pretty_print_values**(*popt*, *pcov*, *params*, *extra_decimal_places=0*)

Takes a numpy array of parameters, the corresponding covariance matrix and a set of parameter names and prints the scaled variables and principal 1-s.d.uncertainties (np.sqrt(pcov[i][i])) and scaling factor in an easy to read format.

Parameters

- **popt** (*numpy array*) – Parameter values
- **pcov** (*2D numpy array*) – Variance-covariance matrix
- **extra_decimal_places** (*int, optional*) – extra precision for values, defaults to 0

burnman.utils.misc.**pretty_print_table**(*table*, *use_tabs=False*)

Takes a 2d table and prints it in a nice text based format. If use_tabs=True then only is used as a separator. This is useful for importing the data into other apps (Excel, ...). The default is to pad the columns with spaces to make them look neat. The first column is left aligned, while the remainder is right aligned.

burnman.utils.misc.**sort_table**(*table*, *col=0*)

Sort the table according to the column number

burnman.utils.misc.**read_table**(*filename*)

burnman.utils.misc.**lookup_and_interpolate**(*table_x*, *table_y*, *x_value*)

burnman.utils.misc.**attribute_function**(*m*, *attributes*, *powers=[]*, *using_volume=False*)

Returns a callable function that evaluates a derived material property defined as a product of material attributes (optionally raised to given powers) at specified thermodynamic conditions.

The returned function is designed to be compatible with *burnman.optimize.nonlinear_fitting.confidence_prediction_bands()*, which expects functions of the form `f(x)` where *x* contains composition and thermodynamic state variables.

The expected order of arguments in *x* depends on the value of *using_volume*: - If `using_volume=False`, the function expects `x = [X1, X2, ..., P, T, _]` and the state of the material is set using *set_state* - If `using_volume=True`, the function expects `x = [X1, X2, ..., V, T, _]` and the state of the material is set using *set_state_with_volume*

The function sets the state of the material using the appropriate method, evaluates the specified attributes, raises them to the corresponding powers, and returns the product.

Parameters

- **m** (*burnman.Material*) – The material instance whose attributes are evaluated.
- **attributes** (*list of str*) – List of material attributes to include in the product expression (e.g., ['K_T', 'rho']).
- **powers** (*list of float*) – List of exponents to raise each corresponding attribute to. If empty, defaults to 1.0 for each attribute.
- **using_volume** (*bool*) – If True, the input function defines the thermodynamic state using volume and temperature (V, T). If False, pressure and temperature (P, T) are used. This argument also determines the expected order of the final three elements in the input vector *x*.

Returns

A function `f(x)` that evaluates the product of the specified attributes, each raised to the corresponding power, at the composition and thermodynamic conditions specified by *x*. The format of *x* is determined by *using_volume*, and is compatible with the *x_array* argument of *burnman.optimize.nonlinear_fitting.confidence_prediction_bands()*.

Return type

function

burnman.utils.misc.**extract_lines_between_markers**(*file*, *start_string*, *end_string*, *inclusive=False*)

Extract lines from a file between two marker strings.

Parameters

- **file** (*str*) – Path to the input text file.
- **start_string** (*str*) – The marker string indicating where to start collecting lines.
- **end_string** (*str*) – The marker string indicating where to stop collecting lines.
- **inclusive** (*bool, optional*) – Whether to include the lines containing start_string and end_string, defaults to False.

Returns

A list of lines between the two markers.

Return type
list[str]

burnman.utils.misc.**run_cli_program_with_input**(*program_path*, *stdin*, *verbose=True*)

Run a command-line program with provided input and capture its output.

Parameters

- **program_path** (*str | list[str]*) – Path to the CLI executable or a list of command-line arguments.
- **stdin** (*str*) – The input string to pass to the program via standard input.
- **verbose** (*bool, optional*) – If True, prints the program's output to stdout, defaults to True.

Returns
The standard output produced by the program.

Return type
str

CHAPTER

TWENTYONE

CHANGES

The following is a list of recent improvements to BurnMan.

CHAPTER

TWENTYTWO

2021-08-02

- BurnMan 0.9.0 is released.

  The BurnMan Team, 2021/08/02

CHAPTER

TWENTYTHREE

2021-08-04

- The BurnMan homepage is updated and moved to https://geodynamics.github.io/burnman/

Bob Myhill and Timo Heister, 2021/08/04

CHAPTER

TWENTYFOUR

2021-08-05

- `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

CHAPTER

TWENTYFIVE

2021-09-27

- 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

CHAPTER

TWENTYSIX

2021-10-03

- 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 [Myh22]. Examples are provided in the file examples/example_anisotropic_mineral.py.

  Bob Myhill, 2021/10/03

CHAPTER

TWENTYSEVEN

2021-10-08

- The experimental function `burnman.equilibrate` now allows users to calculate equilibrium assemblage properties while allowing the bulk composition to vary. An example is provided in the file examples/example_olivine_binary.py.

  Bob Myhill, 2021/10/08

CHAPTER

TWENTYEIGHT

2021-10-22

- The ideal solution model has been updated to allow site multiplicities to vary linearly with endmember proportions. It therefore implements a Temkin-type model [Tem45].
- As a result of the changes to the ideal model, the site_multiplicities attribute of a solid solution is now a 2D array.

  Bob Myhill, 2021/10/22

CHAPTER

TWENTYNINE

2021-10-22

- A new BoundaryLayerPerturbation class has been implemented that allows the user to create a thermal perturbation to a planetary layer according to the model proposed by [RM81]. This perturbation is exponential, taking the form: T = a*exp((r - r1)/(r0 - r1)*c) + b*exp((r - r0)/(r1 - r0)*c). The user defines a, b and c using three related parameters: rayleigh_number (Ra = c^4), temperature_change (the total difference in temperature perturbation across the layer, (a + b)*exp(c)), and boundary_layer_ratio (a/b). Once initialised, a BoundaryLayerPerturbation object can be modified using the function set_model_thermal_gradients, which sets the thermal gradient (dT/dr) at the bottom and top of the layer).

  Bob Myhill, 2021/10/22

CHAPTER

THIRTY

2021-10-22

- A new equation of state, BroshCalphad has been implemented, as described in [BMS07]. This new equation of state is used to define the properties of the BCC, FCC, HCP and liquid polymorphs of iron, as parameterised by [SE15]. These endmembers can be initialised from the BurnMan mineral database (`bcc_iron`, `fcc_iron`, `hcp_iron` and `liquid_iron`).

  Bob Myhill, 2021/10/22

CHAPTER

THIRTYONE

2021-10-30

- The *burnman.Composite* class now has a *chemical_potential* method. This replaces the more limited function `burnman.tools.chemistry.chemical_potentials()`. The related functions *burnman.tools.chemistry.fugacity()* and *burnman.tools.chemistry.relative_fugacity()* have been updated to use this method, and to take Composites as arguments, rather than lists of Minerals.

  Bob Myhill, 2021/10/30

CHAPTER

THIRTYTWO

2021-11-02

- Running the examples now requires BurnMan to be installed. This change is made to ensure that tester behaviour always matches installed behaviour, rather than local behaviour (the cause of the problem with the 1.0.0 release). If the user wishes to develop the BurnMan code, they may install the module from the top-level directory using pip with the *-e* (editable) flag: *python -m pip install -e .*.

  Bob Myhill, 2021/11/02

CHAPTER

THIRTYTHREE

2022-01-28

- BurnMan now has a Calibrant class. This class is a stripped-down version of a Material that is initialised with a params dictionary and a function to compute the pressure or volume at given conditions.

  Objects derived from this class have the ability to output pressure/volume as a function of volume/pressure and temperature. The user can pass a V-T or P-T covariance matrix as an additional optional argument, in which case BurnMan will propagate the errors and output a full P-V-T or V-P-T covariance matrix.

  A helper function is provided to convert from pressure calculated using one calibration into pressure calculated using another calibration of the same material.

  This class is expected to be used by experimental petrologists and those interpreting high pressure experimental data.

  Bob Myhill, 2022/01/28

CHAPTER

THIRTYFOUR

2022-02-05

- New: The EoS fitting functions now have additional (optional) arguments. delta_params allows the user to set initial values for the change in parameters (i.e. initial step size). bounds allows the user to set hard bounds on the values of each of the parameters.

  Bob Myhill, 2022/02/05

CHAPTER

THIRTYFIVE

2022-02-05

- New: BurnMan now has a new fitting function, *burnman.optimize.eos_fitting.fit_XPTp_data()*. It has the same arguments as *burnman.optimize.eos_fitting.fit_PTp_data()*, but instead of fitting data to a mineral of fixed composition, it is able to simultaneously fit parameters of solution model parameters, including equation of state parameters for each endmember.

  Bob Myhill, 2022/02/05

CHAPTER

THIRTYSIX

2022-02-07

- Changed: The default name of the solution class has been changed from `burnman.classes.SolidSolution` to `burnman.classes.Solution`. An alias has been provided for backwards compatibility.

  Bob Myhill, 2022/02/07

CHAPTER

THIRTYSEVEN

2022-02-16

- Changed: BurnMan now has another submodule called utils. Many of the functions that used to be in the tools submodule have now been moved into the utils submodule. Functions residing in utils do not depend on other burnman submodules (in rare cases they depend on objects defined in constants.py). Functions residing in tools are not called by other submodules in BurnMan. Thus, utils is a relatively low level submodule, while tools is a high level submodule. This change has been made to avoid future issues with circular imports.

  Bob Myhill, 2022/02/16

CHAPTER

THIRTYEIGHT

2022-02-16

- New: The BurnMan Material classes now have a method called set_state_with_volume that sets state using volume and temperature rather than pressure and temperature.

  Bob Myhill, 2022/02/16

CHAPTER

THIRTYNINE

2022-06-24

- New: BurnMan now has an ElasticSolution class that implements the elastic solution proposed in [Myh18]. This class defines excess thermodynamic properties of solutions relative to the properties of the solution endmembers at fixed volume and temperature, rather than at fixed pressure and temperature. Thus, the models are most easily expressed through parameterisations of the Helmholtz energy, rather than the Gibbs energy.

  Bob Myhill, 2022/06/24

CHAPTER

FORTY

2022-06-25

- New: BurnMan now has a function method for defining solution/elastic solution models. To use this method, the user should provide a function for the excess nonconfigurational Gibbs/Helmholtz energy during initialisation of the model. The ideal entropy is calculated internally as for all the other solution models.

  Bob Myhill, 2022/06/25

CHAPTER

FORTYONE

2022-10-03

- BurnMan now has two new helper functions: *burnman.tools.chemistry.reactions_from_stoichiometric_matrix()* *burnman.tools.chemistry.reactions_from_formulae()*. These functions generate a complete list of reactions (forward and reverse) from either the stoichiometric matrix (a 2D numpy array containing the amount of component j in phase i), or from a list of formulae (as strings or dictionaries of elemental amounts).

  Bob Myhill, 2022/10/03

CHAPTER

FORTYTWO

2022-10-22

- BurnMan Solution and ElasticSolution objects are now instantiated with a SolutionModel object as a solution_model parameter. The use of the solution_type parameter has been removed completely, along with all of the optional parameters that were originally passed as parameters to SolutionModel.

  Bob Myhill, 2022/10/22

CHAPTER

FORTYTHREE

2022-10-22

- BurnMan now has a generalised PolynomialSolution class. The non-ideal excesses in this model are polynomial functions of composition. The coefficients of these polynomials are either constant internal energy, entropy and volume (i.e. Gxs = (Eijk... - T Sijk... + P Vijk...) xi xj xk...) or a linear sum of Mineral objects (i.e. Gxs = amijk mineralm.gibbs ... xi xj xk ...). Coefficients for both of these forms are passed as a list of lists to instantiate a PolynomialSolution object, which can then be passed as usual to instantiate a Solution object.

  Optionally, a transformation matrix can be passed that allows users to define the coefficients above as functions of a modified set of endmember proportions: p'i = Aij pj. This may be useful when dealing with ordering, as expressing excess terms as a function of order parameter is often much more illuminating than expressing them in terms of endmember proportions.

  This class can deal with arbitrarily high powers in endmember proportions. However, because the class internally converts the list of lists to numpy arrays, high powers of solutions with a large number of endmembers will create very large arrays (with order n_endmembers^(highest power) elements). This may significantly slow down calculations.

  For most purposes, using dense numpy arrays is much faster than using sparse arrays in COO format (a downside of using python). Should users need particularly complex solutions with high power terms, a modified solution model will be required.

  Bob Myhill, 2022/10/22

CHAPTER

FORTYFOUR

2022-10-23

- Four new property modifier formulations are provided, which can be specified with the names “debye”, “debye_delta”, “einstein” and “einstein_delta”. These are based on the Debye and Einstein models of thermal energy. The heat capacity (“debye”, “einstein”) or entropy (“debye_delta”, “einstein_delta”) are based on the heat capacity of the respective thermal model, and so reach a maximum at high temperature. The excess energy, entropy and heat capacity are all zero at 0 K. The excess volume is always zero.

  Bob Myhill, 2022/10/23

CHAPTER

FORTYFIVE

2023-06-20

- BurnMan now has an implementation of the SLB2022 dataset.

  Bob Myhill, 2023/06/20

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.

[ADM+13] Simone Anzellini, A Dewaele, M Mezouar, P Loubeyre, and Guillaume Morard. Melting of iron at Earth's inner core boundary based on fast X-ray diffraction. *Science*, 340(6131):464–466, 2013.

[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.

[Bir47] Francis Birch. Finite elastic strain of cubic crystals. *Physical review*, 71(11):809, 1947.

[BMS07] Eli Brosh, Guy Makov, and Roni Z. Shneck. Application of CALPHAD to high pressures. *Calphad*, 31(2):173–185, 2007. URL: https://www.sciencedirect.com/science/article/pii/S0364591607000168, doi:https://doi.org/10.1016/j.calphad.2006.12.008.

[BS81] JM Brown and TJ Shankland. Thermodynamic parameters in the Earth as determined from seismic profiles. *Geophys. J. Int.*, 66(3):579–596, 1981. URL: http://gji.oxfordjournals.org/content/66/3/579.short.

[BK77] MST Bukowinski and L Knopoff. Physics and chemistry of iron and potassium at lower-mantle and core pressures. In *High-Pressure Research*, pages 367–387. Elsevier, 1977.

[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_2O-CaO-FeO-MgO-Al_2O_3-SiO_2-H_2O-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.

[HS98] Juichiro Hama and Kaichi Suito. Equation of state of $MgSiO_3$ perovskite and its thermoelastic properties under lower mantle conditions. *Journal of Geophysical Research: Solid Earth*, 103(B4):7443–7462, 1998.

[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 K_2O-Na_2O-CaO-MgO-MnO-FeO-Fe_2O_3-Al_2O_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 CO_2 and H_2O 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.

[HP98] TJB Holland and RTJB Powell. An internally consistent thermodynamic data set for phases of petrological interest. *Journal of metamorphic Geology*, 16(3):309–343, 1998.

[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.

[JR96] Ian Jackson and Sally M Rigden. Analysis of PVT data: constraints on the thermoelastic properties of high-pressure minerals. *Physics of the earth and planetary interiors*, 96(2-3):85–112, 1996.

[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.

[Kat22] Tomoo Katsura. A revised adiabatic temperature profile for the mantle. *Journal of Geophysical Research: Solid Earth*, 127(2):e2021JB023562, 2022.

[Kea54] A Keane. An Investigation of Finite Strain in an Isotropic Material Subjected to Hydrostatic Pressure and its Seismological Applications. *Australian Journal of Physics*, 7(2):322, 1954. URL: http://www.publish.csiro.au/?paper=PH540322, doi:10.1071/PH540322.

[KEB95] BLN Kennett, E R Engdahl, and R Buland. Constraints on seismic velocities in the Earth from traveltimes. *Geophys. J. Int.*, 122(1):108–124, 1995. URL: http://gji.oxfordjournals.org/content/122/1/108.short.

[KE91] BLN Kennett and ER Engdahl. Traveltimes for global earthquake location and phase identification. *Geophysical Journal International*, 105(2):429–465, 1991.

[KHM+12] Y Kudo, K Hirose, M Murakami, Y Asahara, H Ozawa, Y Ohishi, and N Hirao. Sound velocity measurements of $CaSiO_3$ perovskite to 133 GPa and implications for lowermost mantle seismic anomalies. *Earth Planet. Sci. Lett.*, 349:1–7, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X1200324X.

[KED08] B Kustowski, G Ekstrom, and AM Dziewonski. Anisotropic shear-wave velocity structure of the Earth's mantle: a global model. *J. Geophys. Res.*, 113(B6):B06306, 2008. URL: http://www.agu.org/pubs/crossref/2008/2007JB005169.shtml.

[LCDR12] V Lekic, S Cottaar, A M Dziewonski, and B Romanowicz. Cluster analysis of global lower mantle tomography: A new class of structure and implications for chemical heterogeneity. *Earth Planet. Sci. Lett.*, 357-358:68–77, 2012. URL: http://www.sciencedirect.com/science/article/pii/S0012821X12005109.

[LvdH08] C Li and RD van der Hilst. A new global model for P wave speed variations in Earth's mantle. *Geochem. Geophys. Geosyst.*, 2008. URL: http://onlinelibrary.wiley.com/doi/10.1029/2007GC001806/full.

[LSMM13] JF Lin, S Speziale, Z Mao, and H Marquardt. Effects of the electronic spin transitions of iron in lower mantle minerals: Implications for deep mantle geophysics and geochemistry. *Rev. Geophys.*, 2013. URL: http://onlinelibrary.wiley.com/doi/10.1002/rog.20010/full.

[LVJ+07] Jung-Fu Lin, György Vankó, Steven D. Jacobsen, Valentin Iota, Viktor V. Struzhkin, Vitali B. Prakapenka, Alexei Kuznetsov, and Choong-Shik Yoo. Spin transition zone in Earth's lower mantle. *Science*, 317(5845):1740–1743, 2007. URL: http://www.sciencemag.org/content/317/5845/1740.abstract, arXiv:http://www.sciencemag.org/content/317/5845/1740.full.pdf.

[LA22] Eduardo Lozano and Tariq D Aslam. A robust three-parameter reference curve for condensed phase materials. *Journal of Applied Physics*, 2022.

[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.

[Mor29] Philip M Morse. Diatomic molecules according to the wave mechanics. II. Vibrational levels. *Physical review*, 34(1):57, 1929.

[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 $MgSiO_3$ 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.

[Myh25a] R Myhill. The SPOCK equation of state for condensed phases under arbitrary compression. *Geophysical Journal International*, 241(2):934–940, 2025.

[Myh18] R. Myhill. The elastic solid solution model for minerals at high pressures and temperatures. *Contributions to Mineralogy and Petrology*, 173(2):12, February 2018. doi:10.1007/s00410-017-1436-z.

[Myh22] R. Myhill. An anisotropic equation of state for high pressure, high temperature applications. *Geophysical Journal International*, May 2022. doi:10.1093/gji/ggac180.

[Myh24] Robert Myhill. An anisotropic equation of state for solid solutions, with application to plagioclase. *Geophysical Journal International*, 239(3):1900–1909, 2024.

[Myh25b] Robert Myhill. A model of elastic softening and displacive phase transitions in anisotropic phases, with application to stishovite and post-stishovite. *Geophysical Journal International*, 241(2):770–796, 2025.

[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.

[PWK+22] Ana-Catalina Plesa, Mark Wieczorek, Martin Knapmeyer, Attilio Rivoldini, Michaela Walterova, and Doris Breuer. Interior dynamics and thermal evolution of Mars–a geodynamic perspective. In *Advances in geophysics*, volume 63, pages 179–230. Elsevier, 2022.

[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.

[PH94] Roger Powell and Tim Holland. Optimal geothermometry and geobarometry. *American Mineralogist*, 79(1-2):120–133, 1994.

[PH99] Roger Powell and Tim Holland. Relating formulations of the thermodynamics of mineral solid solutions; activity modeling of pyroxenes, amphiboles, and micas. *American Mineralogist*, 84(1-2):1–14, 1999. URL: http://ammin.geoscienceworld.org/content/84/1-2/1.abstract, arXiv:http://ammin.geoscienceworld.org/content/84/1-2/1.full.pdf+html.

[RM81] Frank M. Richter and Dan P. McKenzie. Parameterizations for the horizontally averaged temperature of infinite Prandtl number convection. *Journal of Geophysical Research: Solid Earth*, 86(B3):1738–1744, 1981. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/JB086iB03p01738, arXiv:https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/JB086iB03p01738, doi:https://doi.org/10.1029/JB086iB03p01738.

[RDvHW11] J Ritsema, A Deuss, H. J. van Heijst, and J.H. Woodhouse. S40RTS: a degree-40 shear-velocity model for the mantle from new Rayleigh wave dispersion, teleseismic traveltime and normal-mode splitting function. *Geophys. J. Int.*, 184(3):1223–1236, 2011. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2010.04884.x/full.

[SE15] S. Saxena and G. Eriksson. Thermodynamics of Fe-S at ultra-high pressure. *Calphad*, 51:202–205, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0364591615300249, doi:https://doi.org/10.1016/j.calphad.2015.09.009.

[Sch16] F. A. H. Schreinemakers. In-, mono-, and di-variant equilibria. VIII. Further consideration of the bivariant regions; the turning lines. *Proc. K. Akad. Wet. (Netherlands)*, 18:1539–1552, 1916.

[SZN12] BSA Schuberth, C Zaroli, and G Nolet. Synthetic seismograms for a synthetic Earth: long-period P- and S-wave traveltime variations can be explained by temperature alone. *Geophys. J. Int.*, 188(3):1393–1412, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.1365-246X.2011.05333.x/full.

[SDT02] Sang-Heon Shim, Thomas S Duffy, and Kenichi Takemura. Equation of state of gold and its application to the phase boundaries near 660 km depth in Earth's mantle. *Earth and Planetary Science Letters*, 203(2):729–739, 2002.

[SFBG10] NA Simmons, AM Forte, L Boschi, and SP Grand. GyPSuM: A joint tomographic model of mantle density and seismic wave speeds. *J. Geophys. Res.*, 115(B12):B12310, 2010. URL: http://www.agu.org/pubs/crossref/2010/2010JB007631.shtml.

[SMJM12] NA Simmons, SC Myers, G Johanneson, and E Matzel. LLNL-G3Dv3: Global P wave tomography model for improved regional and teleseismic travel time prediction. *J. Geophys. Res.*, 2012. URL: http://onlinelibrary.wiley.com/doi/10.1029/2012JB009525/full.

[SD04] F.D. Stacey and P.M. Davis. High pressure equations of state with applications to the lower mantle and core. *Physics of the Earth and Planetary Interiors*, 142(3-4):137–184, may 2004. URL: http://linkinghub.elsevier.com/retrieve/pii/S0031920104001049, doi:10.1016/j.pepi.2004.02.003.

[Sta77] Frank D. Stacey. A thermal model of the Earth. *Physics of the Earth and Planetary Interiors*, 15(4):341–348, 1977. URL: https://www.sciencedirect.com/science/article/pii/0031920177900966, doi:https://doi.org/10.1016/0031-9201(77)90096-6.

[SD00] Frank D. Stacey and Frank D. The K-primed approach to high-pressure equations of state. *Geophysical Journal International*, 143(3):621–628, dec 2000. URL: https://academic.oup.com/gji/article-lookup/doi/10.1046/j.1365-246X.2000.00253.x, doi:10.1046/j.1365-246X.2000.00253.x.

[SH08] Bernhard Steinberger and R Holme. Mantle flow models with core-mantle boundary constraints and chemical heterogeneities in the lowermost mantle. *Journal of Geophysical Research: Solid Earth*, 2008.

[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.

[SLB22] Lars Stixrude and Carolina Lithgow-Bertelloni. Thermal expansivity, heat capacity and bulk modulus of the mantle. *Geophysical Journal International*, 228(2):1119–1149, 09 2022. URL: https://doi.org/10.1093/gji/ggab394, arXiv:https://academic.oup.com/gji/article-pdf/228/2/1119/40854679/ggab394.pdf, doi:10.1093/gji/ggab394.

[SLB24] Lars Stixrude and Carolina Lithgow-Bertelloni. Thermodynamics of mantle minerals–III: the role of iron. *Geophysical Journal International*, 237(3):1699–1733, 2024.

[SDG11] Elinor Styles, D. Rhodri Davies, and Saskia Goes. Mapping spherical seismic into physical structure: biases from 3-D phase-transition and thermal boundary-layer heterogeneity. *Geophys. J. Int.*, 184(3):1371–1378, March 2011. URL: http://gji.oxfordjournals.org/cgi/doi/10.1111/j.1365-246X.2010.04914.x, doi:10.1111/j.1365-246X.2010.04914.x.

[Sun91] B. Sundman. An assessment of the Fe-O system. *Journal of Phase Equilibria*, 12(2):127–140, 1991. URL: https://doi.org/10.1007/BF02645709, doi:10.1007/BF02645709.

[Tac00] PJ Tackley. Mantle convection and plate tectonics: Toward an integrated physical and chemical theory. *Science*, 288(5473):2002–2007, 2000. URL: http://www.sciencemag.org/content/288/5473/2002.short.

[Tem45] Mo Temkin. Mixtures of fused salts as ionic solutions. *Acta Phys. Chem., USSR*, 20:411, 1945.

[TRCT05] A To, B Romanowicz, Y Capdeville, and N Takeuchi. 3D effects of sharp boundaries at the borders of the African and Pacific Superplumes: Observation and modeling. *Earth Planet. Sci. Lett.*, 233(1-2):1447–1460, 2005.

[TDRY04] Jeannot Trampert, Frédéric Deschamps, Joseph Resovsky, and Dave Yuen. Probabilistic tomography maps chemical heterogeneities throughout the lower mantle. *Science (New York, N.Y.)*, 306(5697):853–6, October 2004. URL: http://www.sciencemag.org/content/306/5697/853.full, doi:10.1126/science.1101996.

[TVV01] Jeannot Trampert, Pierre Vacher, and Nico Vlaar. Sensitivities of seismic velocities to temperature, pressure and composition in the lower mantle. *Phys. Earth Planet. Int.*, 124(3-4):255–267, August 2001. URL: http://www.sciencedirect.com/science/article/pii/S0031920101002011, doi:10.1016/S0031-9201(01)00201-1.

[VSFR87] Pascal Vinet, John R Smith, John Ferrante, and James H Rose. Temperature effects on the universal equation of state of solids. *Physical Review B*, 35(4):1945, 1987. doi:10.1103/PhysRevB.35.1945.

[VFSR86] PJJR Vinet, J Ferrante, JR Smith, and JH Rose. A universal equation of state for solids. *Journal of Physics C: Solid State Physics*, 19(20):L467, 1986. doi:10.1088/0022-3719/19/20/001.

[Wal67] Duane C. Wallace. Thermoelasticity of stressed materials and comparison of various elastic constants. *Physical Review*, 162(3):776–789, 1967. doi:10.1103/PhysRev.162.776.

[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.

[WMTZ15] SP Weppner, JP McKelvey, KD Thielen, and AK Zielinski. A variable polytrope index applied to planet and material models. *Monthly Notices of the Royal Astronomical Society*, 452(2):1375–1393, 2015.

[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 $MgSiO_3$-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

AA (*class in burnman.eos.aa*), 407
abs_line_project() (*in module burnman.optimize.nonlinear_fitting*), 792
activities (*burnman.AnisotropicSolution property*), 513
activities (*burnman.ElasticSolution property*), 280
activities (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
activities (*burnman.RelaxedAnisotropicSolution property*), 550
activities (*burnman.RelaxedSolution property*), 537
activities (*burnman.Solution property*), 270
activities() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
activities() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
activities() (*burnman.classes.solutionmodel.IdealSolution method*), 572
activities() (*burnman.classes.solutionmodel.MechanicalSolution method*), 567
activities() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
activities() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 585
activity_coefficients (*burnman.AnisotropicSolution property*), 513
activity_coefficients (*burnman.ElasticSolution property*), 280
activity_coefficients (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
activity_coefficients (*burnman.RelaxedAnisotropicSolution property*), 550
activity_coefficients (*burnman.RelaxedSolution property*), 537
activity_coefficients (*burnman.Solution property*), 270
activity_coefficients() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
activity_coefficients() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
activity_coefficients() (*burnman.classes.solutionmodel.IdealSolution method*), 571
activity_coefficients() (*burnman.classes.solutionmodel.MechanicalSolution method*), 567
activity_coefficients() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
activity_coefficients() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 585
add_components() (*burnman.Composition method*), 608
adiabatic_profile() (*in module burnman.classes.geotherm*), 629
AdiabaticGeotherm (*class in burnman.classes.geotherm*), 628
AK135 (*class in burnman.classes.seismic*), 681
aKT() (*in module burn-*

man.eos.bukowinski_electronic), 480
alpha (*burnman.AnisotropicMaterial property*), 301
alpha (*burnman.AnisotropicMineral property*), 318
alpha (*burnman.AnisotropicSolution property*), 513
alpha (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
alpha (*burnman.CombinedMineral property*), 263
alpha (*burnman.Composite property*), 337
alpha (*burnman.ElasticSolution property*), 283
alpha (*burnman.Layer property*), 637
alpha (*burnman.Material property*), 234
alpha (*burnman.Mineral property*), 256
alpha (*burnman.optimize.composition_fitting.DummyComposition property*), 772
alpha (*burnman.PerplexMaterial property*), 243
alpha (*burnman.Planet property*), 646
alpha (*burnman.RelaxedAnisotropicSolution property*), 550
alpha (*burnman.RelaxedComposite property*), 344
alpha (*burnman.RelaxedSolution property*), 537
alpha (*burnman.Solution property*), 274
alphaV_excess() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
alphaV_excess() (*burnman.classes.solutionmodel.FunctionSolution method*), 600
alphaV_excess() (*burnman.classes.solutionmodel.IdealSolution method*), 572
alphaV_excess() (*burnman.classes.solutionmodel.MechanicalSolution method*), 567
alphaV_excess() (*burnman.classes.solutionmodel.SubregularSolution method*), 594
alphaV_excess() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 585
alphaV_excess() (*burnman.SolutionModel method*), 531
Anderson (*class in burnman.classes.geotherm*), 624
AnharmonicDebye (*class in burnman.eos.anharmonic_debye*), 475
AnharmonicPrefactorModel (*class in burnman.eos.anharmonic_prefactor_models*), 476
AnharmonicThermalModelBase (*class in burnman.eos.anharmonic_thermal_models*), 477
AnisotropicMaterial (*class in burnman*), 297
AnisotropicMineral (*class in burnman*), 311
AnisotropicSolution (*class in burnman*), 511
Anzellini2013 (*class in burnman.classes.geotherm*), 627
append() (*burnman.optimize.nonlinear_solvers.Iterates method*), 798
array_to_rational_matrix() (*in module burnman.utils.math*), 839
assemblage_affinity_covariance_matrix() (*in module burnman.tools.thermobarometry*), 810
assemblage_affinity_misfit() (*in module burnman.tools.thermobarometry*), 811
assemblage_set_state_from_params() (*in module burnman.tools.thermobarometry*), 810
assemblage_state_misfit() (*in module burnman.tools.thermobarometry*), 811
AsymmetricRegularSolution (*class in burnman.classes.solutionmodel*), 577
atomic_composition (*burnman.Composition property*), 609
atomic_masses (*in module burnman.utils.chemistry*), 828
attenuation_correction() (*in module burnman.tools.seismic*), 821
attribute_function() (*in module burnman.utils.misc*), 843
average_bulk_moduli() (*burnman.averaging_schemes.AveragingScheme method*), 616
average_bulk_moduli() (*burnman.averaging_schemes.HashinShtrikmanAverage method*), 622
average_bulk_moduli() (*burnman.averaging_schemes.HashinShtrikmanLower method*), 622
average_bulk_moduli() (*burnman.averaging_schemes.HashinShtrikmanUpper method*), 621
average_bulk_moduli() (*burnman.averaging_schemes.Reuss method*), 618
average_bulk_moduli() (*burnman.averaging_schemes.Voigt method*),

617
average_bulk_moduli() (*burnman.averaging_schemes.VoigtReussHill method*), 619
average_density (*burnman.Planet property*), 640
average_shear_moduli() (*burnman.averaging_schemes.AveragingScheme method*), 617
average_shear_moduli() (*burnman.averaging_schemes.HashinShtrikmanAverage method*), 623
average_shear_moduli() (*burnman.averaging_schemes.HashinShtrikmanLower method*), 622
average_shear_moduli() (*burnman.averaging_schemes.HashinShtrikmanUpper method*), 621
average_shear_moduli() (*burnman.averaging_schemes.Reuss method*), 619
average_shear_moduli() (*burnman.averaging_schemes.Voigt method*), 618
average_shear_moduli() (*burnman.averaging_schemes.VoigtReussHill method*), 620
AveragingScheme (*class in burnman.averaging_schemes*), 616
axisem_formatted_data_and_reference() (*in module burnman.tools.output_seismo*), 822

B

beta_S (*burnman.AnisotropicMaterial property*), 301
beta_S (*burnman.AnisotropicMineral property*), 314
beta_S (*burnman.AnisotropicSolution property*), 513
beta_S (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
beta_S (*burnman.CombinedMineral property*), 263
beta_S (*burnman.Composite property*), 337
beta_S (*burnman.ElasticSolution property*), 283
beta_S (*burnman.Layer property*), 637
beta_S (*burnman.Material property*), 234
beta_S (*burnman.Mineral property*), 256
beta_S (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
beta_S (*burnman.PerplexMaterial property*), 243
beta_S (*burnman.Planet property*), 646
beta_S (*burnman.RelaxedAnisotropicSolution property*), 550
beta_S (*burnman.RelaxedComposite property*), 344
beta_S (*burnman.RelaxedSolution property*), 537
beta_S (*burnman.Solution property*), 274
beta_T (*burnman.AnisotropicMaterial property*), 301
beta_T (*burnman.AnisotropicMineral property*), 314
beta_T (*burnman.AnisotropicSolution property*), 513
beta_T (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
beta_T (*burnman.CombinedMineral property*), 263
beta_T (*burnman.Composite property*), 337
beta_T (*burnman.ElasticSolution property*), 283
beta_T (*burnman.Layer property*), 637
beta_T (*burnman.Material property*), 234
beta_T (*burnman.Mineral property*), 256
beta_T (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
beta_T (*burnman.PerplexMaterial property*), 243
beta_T (*burnman.Planet property*), 645
beta_T (*burnman.RelaxedAnisotropicSolution property*), 550
beta_T (*burnman.RelaxedComposite property*), 344
beta_T (*burnman.RelaxedSolution property*), 537
beta_T (*burnman.Solution property*), 274
BirchMurnaghanBase (*class in burnman.eos.birch_murnaghan*), 372
BM3 (*class in burnman.eos.birch_murnaghan*), 359
BM3Shear2 (*class in burnman.eos.birch_murnaghan*), 364
BM4 (*class in burnman.eos.birch_murnaghan*), 368
BoundaryLayerPerturbation (*class in burnman*), 637
bracket() (*in module burnman.utils.math*), 835
bragg_williams_excesses() (*in module burnman.eos.property_modifiers*), 483
BroshCalphad (*class in burnman.eos.brosh_calphad*), 411
BrownShankland (*class in burnman.classes.geotherm*), 624
brunt_vasala (*burnman.Layer property*), 632
brunt_vasala (*burnman.Planet property*), 640

bulk_modulus() (*in module burnman.eos.modified_tait*), 482
bulk_modulus() (*in module burnman.eos.morse_potential*), 483
bulk_modulus() (*in module burnman.eos.murnaghan*), 483
bulk_modulus() (*in module burnman.eos.reciprocal_kprime*), 489
bulk_modulus_fourth_order() (*in module burnman.eos.birch_murnaghan*), 478
bulk_modulus_third_order() (*in module burnman.eos.birch_murnaghan*), 478
bulk_modulus_vinet() (*in module burnman.eos.vinet*), 490
bulk_sound_velocity (*burnman.AnisotropicMaterial property*), 301
bulk_sound_velocity (*burnman.AnisotropicMineral property*), 318
bulk_sound_velocity (*burnman.AnisotropicSolution property*), 513
bulk_sound_velocity (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
bulk_sound_velocity (*burnman.CombinedMineral property*), 262
bulk_sound_velocity (*burnman.Composite property*), 334
bulk_sound_velocity (*burnman.ElasticSolution property*), 281
bulk_sound_velocity (*burnman.Layer property*), 635
bulk_sound_velocity (*burnman.Material property*), 231
bulk_sound_velocity (*burnman.Mineral property*), 255
bulk_sound_velocity (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
bulk_sound_velocity (*burnman.PerplexMaterial property*), 238
bulk_sound_velocity (*burnman.Planet property*), 644
bulk_sound_velocity (*burnman.RelaxedAnisotropicSolution property*), 550
bulk_sound_velocity (*burnman.RelaxedComposite property*), 344
bulk_sound_velocity (*burnman.RelaxedSolution property*), 537
bulk_sound_velocity (*burnman.Solution property*), 271
bullen (*burnman.Layer property*), 632
bullen (*burnman.Planet property*), 640
bullen() (*burnman.classes.seismic.AK135 method*), 682
bullen() (*burnman.classes.seismic.Fast method*), 673
bullen() (*burnman.classes.seismic.IASP91 method*), 679
bullen() (*burnman.classes.seismic.PREM method*), 666
bullen() (*burnman.classes.seismic.SeismicTable method*), 664
bullen() (*burnman.classes.seismic.Slow method*), 670
bullen() (*burnman.classes.seismic.STW105 method*), 676
burnman
    module, 1
burnman.calibrants
    module, 762
burnman.classes.geotherm
    module, 623
burnman.minerals
    module, 685
burnman.optimize.composition_fitting
    module, 771
burnman.optimize.eos_fitting
    module, 783
burnman.optimize.linear_fitting
    module, 790
burnman.optimize.nonlinear_fitting
    module, 791
burnman.optimize.nonlinear_solvers
    module, 798
burnman.tools.chemistry
    module, 816
burnman.tools.eos
    module, 813
burnman.tools.equilibration
    module, 802
burnman.tools.output_seismo
    module, 821
burnman.tools.partitioning
    module, 819

burnman.tools.plot
    module, 820
burnman.tools.polytope
    module, 823
burnman.tools.seismic
    module, 821
burnman.tools.solution
    module, 815
burnman.tools.thermobarometry
    module, 809
burnman.tools.unitcell
    module, 802
burnman.utils.anisotropy
    module, 833
burnman.utils.chemistry
    module, 828
burnman.utils.math
    module, 835
burnman.utils.misc
    module, 840
burnman.utils.plotting
    module, 834
burnman.utils.reductions
    module, 840
burnman.utils.unitcell
    module, 826

C

C_p (*burnman.AnisotropicMaterial property*), 300
C_p (*burnman.AnisotropicMineral property*), 317
C_p (*burnman.AnisotropicSolution property*), 512
C_p (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
C_p (*burnman.CombinedMineral property*), 262
C_p (*burnman.Composite property*), 337
C_p (*burnman.ElasticSolution property*), 282
C_p (*burnman.Layer property*), 637
C_p (*burnman.Material property*), 235
C_p (*burnman.Mineral property*), 256
C_p (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
C_p (*burnman.PerplexMaterial property*), 242
C_p (*burnman.Planet property*), 646
C_p (*burnman.RelaxedAnisotropicSolution property*), 549
C_p (*burnman.RelaxedComposite property*), 343
C_p (*burnman.RelaxedSolution property*), 536
C_p (*burnman.Solution property*), 273
C_v (*burnman.AnisotropicMaterial property*), 300
C_v (*burnman.AnisotropicMineral property*), 318
C_v (*burnman.AnisotropicSolution property*), 512
C_v (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
C_v (*burnman.CombinedMineral property*), 262
C_v (*burnman.Composite property*), 337
C_v (*burnman.ElasticSolution property*), 282
C_v (*burnman.Layer property*), 637
C_v (*burnman.Material property*), 234
C_v (*burnman.Mineral property*), 256
C_v (*burnman.optimize.composition_fitting.DummyCompositionSolu property*), 771
C_v (*burnman.PerplexMaterial property*), 242
C_v (*burnman.Planet property*), 646
C_v (*burnman.RelaxedAnisotropicSolution property*), 549
C_v (*burnman.RelaxedComposite property*), 343
C_v (*burnman.RelaxedSolution property*), 536
C_v (*burnman.Solution property*), 273
calc_shear_velocities() (*in module contrib.CHRU2014.paper_fit_data*), 220
calculate_constraints() (*in module burnman.tools.equilibration*), 802
calculate_jacobian() (*in module burnman.optimize.nonlinear_fitting*), 793
calculate_nakajima_fp_pv_partition_coefficient() (*in module burnman.tools.partitioning*), 819
calculate_property_modifications() (*in module burnman.eos.property_modifiers*), 484
calculate_transformed_parameters() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
Calibrant (*class in burnman*), 352
Cell (*class in burnman.utils.plotting*), 834
cell_parameters (*burnman.AnisotropicMineral property*), 313
cell_parameters (*burnman.AnisotropicSolution property*), 513
cell_parameters (*burnman.RelaxedAnisotropicSolution property*), 550
cell_parameters_to_vectors() (*in module burnman*), 329
cell_parameters_to_vectors() (*in module burnman.utils.unitcell*), 826

cell_vectors (*burnman.AnisotropicMineral property*), 313
cell_vectors (*burnman.AnisotropicSolution property*), 514
cell_vectors (*burnman.RelaxedAnisotropicSolution property*), 551
cell_vectors_to_parameters() (*in module burnman*), 330
cell_vectors_to_parameters() (*in module burnman.utils.unitcell*), 827
change_component_set() (*burnman.Composition method*), 608
check_anisotropic_eos_consistency() (*in module burnman.tools.eos*), 813
check_eos_consistency() (*in module burnman.tools.eos*), 813
check_hyperelastic_eos_consistency() (*in module burnman.tools.eos*), 814
check_standard_parameters() (*burnman.AnisotropicMineral method*), 329
check_standard_parameters() (*burnman.AnisotropicSolution method*), 514
check_standard_parameters() (*burnman.RelaxedAnisotropicSolution method*), 551
chemical_potential() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 289
chemical_potential() (*burnman.Composite method*), 335
chemical_potential() (*burnman.RelaxedComposite method*), 344
chisqr_profile() (*in module burnman.utils.math*), 838
chisqr_profiles() (*in module burnman.utils.math*), 837
christoffel_tensor() (*burnman.AnisotropicMaterial method*), 299
christoffel_tensor() (*burnman.AnisotropicMineral method*), 318
christoffel_tensor() (*burnman.AnisotropicSolution method*), 514
christoffel_tensor() (*burnman.RelaxedAnisotropicSolution method*), 551
clear() (*burnman.utils.misc.OrderedCounter method*), 840
close() (*burnman.optimize.nonlinear_solvers.Iterates method*), 798
closest_point_on_polygon() (*in module burnman.utils.plotting*), 834
closest_point_on_segment() (*in module burnman.utils.plotting*), 834
code (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
CombinedMineral (*class in burnman*), 260
complete_basis() (*in module burnman.utils.math*), 839
Composite (*class in burnman*), 331
composite_polytope_at_constrained_composition() (*in module burnman.tools.polytope*), 824
Composition (*class in burnman*), 607
composition() (*burnman.Composition method*), 609
compositional_array() (*in module burnman.utils.chemistry*), 831
compositional_basis (*burnman.AnisotropicSolution property*), 514
compositional_basis (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 772
compositional_basis (*burnman.RelaxedAnisotropicSolution property*), 551
compositional_basis (*burnman.RelaxedSolution property*), 537
compositional_basis (*burnman.Solution property*), 272
compositional_null_basis (*burnman.AnisotropicSolution property*), 515
compositional_null_basis (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
compositional_null_basis (*burnman.Composite property*), 336
compositional_null_basis (*burnman.ElasticSolution property*), 282
compositional_null_basis (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 772
compositional_null_basis (*burnman.RelaxedAnisotropicSolution property*), 552
compositional_null_basis (*burnman.RelaxedComposite property*), 344

compositional_null_basis (*burnman.RelaxedSolution property*), 538
compositional_null_basis (*burnman.Solution property*), 273
compute_base_properties() (*burnman.AnisotropicSolution method*), 512
compute_base_properties() (*burnman.RelaxedAnisotropicSolution method*), 552
confidence_prediction_bands() (*in module burnman.optimize.nonlinear_fitting*), 795
configurational_entropy() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 583
configurational_entropy() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 605
configurational_entropy() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
configurational_entropy() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
configurational_entropy() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 589
configurational_entropy() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 580
configurational_entropy() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
configurational_entropy() (*burnman.classes.solutionmodel.IdealSolution method*), 571
configurational_entropy() (*burnman.classes.solutionmodel.SubregularSolution method*), 594
configurational_entropy() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 586
CONSTRAINT_VIOLATION (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
contract_compliances() (*in module burnman.utils.anisotropy*), 833
contract_stiffnesses() (*in module burnman.utils.anisotropy*), 833
contract_strains() (*in module burnman.utils.anisotropy*), 833
contract_stresses() (*in module burnman.utils.anisotropy*), 833
contrib.CHRU2014.paper_averaging module, 220
contrib.CHRU2014.paper_benchmark module, 220
contrib.CHRU2014.paper_fit_data module, 220
contrib.CHRU2014.paper_incorrect_averaging module, 220
contrib.CHRU2014.paper_onefit module, 220
contrib.CHRU2014.paper_opt_pv module, 220
contrib.CHRU2014.paper_uncertain module, 221
contrib.cider_tutorial_2014.step_1 module, 57
contrib.cider_tutorial_2014.step_2 module, 58
contrib.cider_tutorial_2014.step_3 module, 59
convert_formula() (*in module burnman.utils.chemistry*), 829
convert_fractions() (*in module burnman.utils.chemistry*), 832
copy() (*burnman.AnisotropicMaterial method*), 301
copy() (*burnman.AnisotropicMineral method*), 319
copy() (*burnman.AnisotropicSolution method*), 515
copy() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 289
copy() (*burnman.CombinedMineral method*), 263
copy() (*burnman.Composite method*), 337
copy() (*burnman.ElasticSolution method*), 283
copy() (*burnman.Material method*), 224
copy() (*burnman.Mineral method*), 257
copy() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 773
copy() (*burnman.PerplexMaterial method*), 243
copy() (*burnman.RelaxedAnisotropicSolution method*), 552
copy() (*burnman.RelaxedComposite method*), 344
copy() (*burnman.RelaxedSolution method*), 538
copy() (*burnman.Solution method*), 274

copy() (*burnman.utils.misc.OrderedCounter method*), 840
copy_documentation() (*in module burnman.utils.misc*), 842
copy_state_from_unrelaxed() (*burnman.RelaxedComposite method*), 343
CORK (*class in burnman.eos.cork*), 416
corner_plot() (*in module burnman.optimize.nonlinear_fitting*), 796
Cp_excess() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
Cp_excess() (*burnman.classes.solutionmodel.FunctionSolution method*), 600
Cp_excess() (*burnman.classes.solutionmodel.IdealSolution method*), 572
Cp_excess() (*burnman.classes.solutionmodel.MechanicalSolution method*), 567
Cp_excess() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
Cp_excess() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 585
Cp_excess() (*burnman.SolutionModel method*), 531
CVoverT() (*in module burnman.eos.bukowinski_electronic*), 480

D

d2helmholtz_dTheta2() (*in module burnman.eos.debye*), 481
d2helmholtz_dTheta2() (*in module burnman.eos.einstein*), 482
damped_newton_solve() (*in module burnman.optimize.nonlinear_solvers*), 801
DampedNewtonSolver (*class in burnman.optimize.nonlinear_solvers*), 800
debug_print() (*burnman.AnisotropicMaterial method*), 301
debug_print() (*burnman.AnisotropicMineral method*), 319
debug_print() (*burnman.AnisotropicSolution method*), 515
debug_print() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 288
debug_print() (*burnman.CombinedMineral method*), 263
debug_print() (*burnman.Composite method*), 332
debug_print() (*burnman.ElasticSolution method*), 283
debug_print() (*burnman.Material method*), 223
debug_print() (*burnman.Mineral method*), 248
debug_print() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 773
debug_print() (*burnman.PerplexMaterial method*), 243
debug_print() (*burnman.RelaxedAnisotropicSolution method*), 552
debug_print() (*burnman.RelaxedComposite method*), 344
debug_print() (*burnman.RelaxedSolution method*), 538
debug_print() (*burnman.Solution method*), 274
debye_delta_excesses() (*in module burnman.eos.property_modifiers*), 484
debye_excesses() (*in module burnman.eos.property_modifiers*), 485
debye_fn() (*in module burnman.eos.debye*), 481
debye_fn_cheb() (*in module burnman.eos.debye*), 481
DebyeTemperatureModelBase (*class in burnman.eos.debye_temperature_models*), 478
default_F_tolerances() (*in module burnman.tools.equilibration*), 803
default_mle_tolerances() (*in module burnman.optimize.eos_fitting*), 783
deformation_experiment() (*in module examples.example_hyperelastic_stress_rates*), 190
deformation_gradient_tensor (*burnman.AnisotropicMineral property*), 312
deformation_gradient_tensor (*burnman.AnisotropicSolution property*), 515
deformation_gradient_tensor (*burnman.RelaxedAnisotropicSolution property*), 552
density (*burnman.AnisotropicMaterial property*),

297
density (*burnman.AnisotropicMineral property*), 319
density (*burnman.AnisotropicSolution property*), 515
density (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
density (*burnman.CombinedMineral property*), 261
density (*burnman.Composite property*), 333
density (*burnman.ElasticSolution property*), 281
density (*burnman.Layer property*), 633
density (*burnman.Material property*), 228
density (*burnman.Mineral property*), 252
density (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 773
density (*burnman.PerplexMaterial property*), 240
density (*burnman.Planet property*), 642
density (*burnman.RelaxedAnisotropicSolution property*), 552
density (*burnman.RelaxedComposite property*), 345
density (*burnman.RelaxedSolution property*), 538
density (*burnman.Solution property*), 271
density() (*burnman.classes.seismic.AK135 method*), 682
density() (*burnman.classes.seismic.Fast method*), 673
density() (*burnman.classes.seismic.IASP91 method*), 679
density() (*burnman.classes.seismic.PREM method*), 666
density() (*burnman.classes.seismic.Seismic1DModel method*), 661
density() (*burnman.classes.seismic.SeismicTable method*), 664
density() (*burnman.classes.seismic.Slow method*), 670
density() (*burnman.classes.seismic.STW105 method*), 676
density() (*burnman.eos.aa.AA method*), 409
density() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
density() (*burnman.eos.birch_murnaghan.BM3 method*), 360
density() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 364
density() (*burnman.eos.birch_murnaghan.BM4 method*), 369
density() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
density() (*burnman.eos.cork.CORK method*), 417
density() (*burnman.eos.dks_liquid.DKS_L method*), 421
density() (*burnman.eos.dks_solid.DKS_S method*), 425
density() (*burnman.eos.EquationOfState method*), 402
density() (*burnman.eos.hp.HP98 method*), 428
density() (*burnman.eos.hp.HP_TMT method*), 433
density() (*burnman.eos.hp.HP_TMTL method*), 436
density() (*burnman.eos.IsothermalEquationOfState method*), 355
density() (*burnman.eos.macaw.MACAW method*), 376
density() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 439
density() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 444
density() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 448
density() (*burnman.eos.modified_tait.MT method*), 380
density() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
density() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
density() (*burnman.eos.morse_potential.Morse method*), 383
density() (*burnman.eos.murnaghan.Murnaghan method*), 387
density() (*burnman.eos.reciprocal_kprime.RKprime method*), 391

density() (*burnman.eos.slb.SLB2 method*), 460
density() (*burnman.eos.slb.SLB3 method*), 465
density() (*burnman.eos.slb.SLB3Conductive method*), 469
density() (*burnman.eos.slb.SLBBase method*), 473
density() (*burnman.eos.spock.SPOCK method*), 394
density() (*burnman.eos.vinet.Vinet method*), 398
dentropy_dTheta() (*in module burnman.eos.debye*), 481
dentropy_dTheta() (*in module burnman.eos.einstein*), 482
dependent_element_indices (*burnman.AnisotropicSolution property*), 515
dependent_element_indices (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
dependent_element_indices (*burnman.Composite property*), 336
dependent_element_indices (*burnman.ElasticSolution property*), 282
dependent_element_indices (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 773
dependent_element_indices (*burnman.RelaxedAnisotropicSolution property*), 553
dependent_element_indices (*burnman.RelaxedComposite property*), 345
dependent_element_indices (*burnman.RelaxedSolution property*), 538
dependent_element_indices (*burnman.Solution property*), 273
depsdn_fixed_VT (*burnman.AnisotropicSolution property*), 512
depsdn_fixed_VT (*burnman.RelaxedAnisotropicSolution property*), 553
depth (*burnman.Planet property*), 640
depth() (*burnman.classes.seismic.AK135 method*), 683
depth() (*burnman.classes.seismic.Fast method*), 673
depth() (*burnman.classes.seismic.IASP91 method*), 679
depth() (*burnman.classes.seismic.PREM method*), 667
depth() (*burnman.classes.seismic.Seismic1DModel method*), 662
depth() (*burnman.classes.seismic.SeismicTable method*), 664
depth() (*burnman.classes.seismic.Slow method*), 670
depth() (*burnman.classes.seismic.STW105 method*), 676
dhelmholtz_dTheta() (*in module burnman.eos.debye*), 481
dhelmholtz_dTheta() (*in module burnman.eos.einstein*), 482
dictionarize_formula() (*in module burnman.utils.chemistry*), 828
DKS_L (*class in burnman.eos.dks_liquid*), 420
DKS_S (*class in burnman.eos.dks_solid*), 424
dmolar_heat_capacity_v_dT() (*in module burnman.eos.debye*), 481
dmolar_heat_capacity_v_dT() (*in module burnman.eos.einstein*), 482
dqdz_relaxed (*burnman.RelaxedAnisotropicSolution property*), 546
dqdz_relaxed (*burnman.RelaxedComposite property*), 343
dqdz_relaxed (*burnman.RelaxedSolution property*), 535
dSdV() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
dTdr() (*burnman.BoundaryLayerPerturbation method*), 638
DummyCompositionSolution (*class in burnman.optimize.composition_fitting*), 771
dvolume_dP() (*burnman.eos.brosh_calphad.BroshCalphad method*), 412
dVrel() (*burnman.eos.anharmonic_prefactor_models.AnharmonicP method*), 476
dVrel() (*burnman.eos.anharmonic_prefactor_models.PowerLaw method*), 476
dVrel() (*burnman.eos.anharmonic_prefactor_models.Sigmoid method*), 477
dVrel() (*burnman.eos.debye_temperature_models.DebyeTemperatu method*), 478
dVrel2() (*burnman.eos.anharmonic_prefactor_models.Anharmonic method*), 476
dVrel2() (*burnman.eos.anharmonic_prefactor_models.PowerLaw method*), 476

dVrel2() (*burnman.eos.anharmonic_prefactor_models.Sigmoid method*), 477
dVrel2() (*burnman.eos.debye_temperature_models.DebyeTemperatureCombinedMineral method*), 478

E

effective_isentropic_bulk_modulus (*burnman.AnisotropicMaterial property*), 301
effective_isentropic_bulk_modulus (*burnman.AnisotropicMineral property*), 319
effective_isentropic_bulk_modulus (*burnman.AnisotropicSolution property*), 516
effective_isentropic_bulk_modulus (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
effective_isentropic_bulk_modulus (*burnman.CombinedMineral property*), 263
effective_isentropic_bulk_modulus (*burnman.Composite property*), 334
effective_isentropic_bulk_modulus (*burnman.ElasticSolution property*), 283
effective_isentropic_bulk_modulus (*burnman.Material property*), 230
effective_isentropic_bulk_modulus (*burnman.Mineral property*), 257
effective_isentropic_bulk_modulus (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 773
effective_isentropic_bulk_modulus (*burnman.PerplexMaterial property*), 243
effective_isentropic_bulk_modulus (*burnman.RelaxedAnisotropicSolution property*), 553
effective_isentropic_bulk_modulus (*burnman.RelaxedComposite property*), 345
effective_isentropic_bulk_modulus (*burnman.RelaxedSolution property*), 538
effective_isentropic_bulk_modulus (*burnman.Solution property*), 274
effective_shear_modulus (*burnman.AnisotropicMaterial property*), 302
effective_shear_modulus (*burnman.AnisotropicMineral property*), 320
effective_shear_modulus (*burnman.AnisotropicSolution property*), 516
effective_shear_modulus (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
effective_shear_modulus (*burnman.CombinedMineral property*), 263
effective_shear_modulus (*burnman.Composite property*), 334
effective_shear_modulus (*burnman.ElasticSolution property*), 284
effective_shear_modulus (*burnman.Material property*), 231
effective_shear_modulus (*burnman.Mineral property*), 257
effective_shear_modulus (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 773
effective_shear_modulus (*burnman.PerplexMaterial property*), 243
effective_shear_modulus (*burnman.RelaxedAnisotropicSolution property*), 553
effective_shear_modulus (*burnman.RelaxedComposite property*), 345
effective_shear_modulus (*burnman.RelaxedSolution property*), 538
effective_shear_modulus (*burnman.Solution property*), 275
einstein_delta_excesses() (*in module burnman.eos.property_modifiers*), 485
einstein_excesses() (*in module burnman.eos.property_modifiers*), 486
ElasticAsymmetricRegularSolution (*class in burnman.classes.elasticsolutionmodel*), 581
ElasticFunctionSolution (*class in burnman.classes.elasticsolutionmodel*), 603
ElasticIdealSolution (*class in burnman.classes.elasticsolutionmodel*), 574
ElasticMechanicalSolution (*class in burnman.classes.elasticsolutionmodel*), 568
ElasticSolidSolution (*in module burnman*), 287
ElasticSolution (*class in burnman*), 279
ElasticSolutionModel (*class in burnman*), 532
ElasticSubregularSolution (*class in burnman.classes.elasticsolutionmodel*), 596
ElasticSymmetricRegularSolution (*class in burnman.classes.elasticsolutionmodel*), 589
elements (*burnman.AnisotropicSolution property*), 516

elements (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
elements (*burnman.Composite property*), 336
elements (*burnman.ElasticSolution property*), 282
elements (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
elements (*burnman.RelaxedAnisotropicSolution property*), 553
elements (*burnman.RelaxedComposite property*), 345
elements (*burnman.RelaxedSolution property*), 539
elements (*burnman.Solution property*), 273
elements() (*burnman.utils.misc.OrderedCounter method*), 840
endmember_formulae (*burnman.AnisotropicSolution property*), 516
endmember_formulae (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 289
endmember_formulae (*burnman.Composite property*), 336
endmember_formulae (*burnman.ElasticSolution property*), 282
endmember_formulae (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
endmember_formulae (*burnman.RelaxedAnisotropicSolution property*), 553
endmember_formulae (*burnman.RelaxedComposite property*), 345
endmember_formulae (*burnman.RelaxedSolution property*), 539
endmember_formulae (*burnman.Solution property*), 273
endmember_names (*burnman.AnisotropicSolution property*), 516
endmember_names (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
endmember_names (*burnman.Composite property*), 336
endmember_names (*burnman.ElasticSolution property*), 282
endmember_names (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
endmember_names (*burnman.RelaxedAnisotropicSolution property*), 553
endmember_names (*burnman.RelaxedComposite property*), 345
endmember_names (*burnman.RelaxedSolution property*), 539
endmember_names (*burnman.Solution property*), 273
endmember_occupancies (*burnman.MaterialPolytope property*), 612
endmember_partial_entropies (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
endmember_partial_entropies (*burnman.Composite property*), 335
endmember_partial_entropies (*burnman.RelaxedComposite property*), 345
endmember_partial_gibbs (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
endmember_partial_gibbs (*burnman.Composite property*), 335
endmember_partial_gibbs (*burnman.RelaxedComposite property*), 345
endmember_partial_volumes (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
endmember_partial_volumes (*burnman.Composite property*), 335
endmember_partial_volumes (*burnman.RelaxedComposite property*), 345
endmembers (*burnman.AnisotropicSolution property*), 516
endmembers (*burnman.ElasticSolution property*), 279
endmembers (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
endmembers (*burnman.RelaxedAnisotropicSolution property*), 553
endmembers (*burnman.RelaxedSolution property*), 539
endmembers (*burnman.Solution property*), 269
endmembers_as_independent_endmember_amounts (*burnman.MaterialPolytope property*), 613
endmembers_per_phase (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290

endmembers_per_phase (*burnman.Composite property*), 336
endmembers_per_phase (*burnman.RelaxedComposite property*), 345
energy (*burnman.Layer property*), 636
energy (*burnman.Planet property*), 645
energy() (*in module burnman.eos.murnaghan*), 483
enthalpy (*burnman.AnisotropicMaterial property*), 302
enthalpy (*burnman.AnisotropicMineral property*), 320
enthalpy (*burnman.AnisotropicSolution property*), 516
enthalpy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
enthalpy (*burnman.CombinedMineral property*), 263
enthalpy (*burnman.Composite property*), 337
enthalpy (*burnman.ElasticSolution property*), 284
enthalpy (*burnman.Material property*), 233
enthalpy (*burnman.Mineral property*), 257
enthalpy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
enthalpy (*burnman.PerplexMaterial property*), 244
enthalpy (*burnman.RelaxedAnisotropicSolution property*), 553
enthalpy (*burnman.RelaxedComposite property*), 345
enthalpy (*burnman.RelaxedSolution property*), 539
enthalpy (*burnman.Solution property*), 275
enthalpy() (*burnman.eos.aa.AA method*), 409
enthalpy() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 373
enthalpy() (*burnman.eos.birch_murnaghan.BM3 method*), 361
enthalpy() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 365
enthalpy() (*burnman.eos.birch_murnaghan.BM4 method*), 369
enthalpy() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
enthalpy() (*burnman.eos.cork.CORK method*), 417
enthalpy() (*burnman.eos.dks_liquid.DKS_L method*), 422
enthalpy() (*burnman.eos.dks_solid.DKS_S method*), 425
enthalpy() (*burnman.eos.EquationOfState method*), 406
enthalpy() (*burnman.eos.hp.HP98 method*), 429
enthalpy() (*burnman.eos.hp.HP_TMT method*), 433
enthalpy() (*burnman.eos.hp.HP_TMTL method*), 437
enthalpy() (*burnman.eos.IsothermalEquationOfState method*), 355
enthalpy() (*burnman.eos.macaw.MACAW method*), 376
enthalpy() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 440
enthalpy() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 444
enthalpy() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 448
enthalpy() (*burnman.eos.modified_tait.MT method*), 380
enthalpy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
enthalpy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
enthalpy() (*burnman.eos.morse_potential.Morse method*), 384
enthalpy() (*burnman.eos.murnaghan.Murnaghan method*), 387
enthalpy() (*burnman.eos.reciprocal_kprime.RKprime method*), 391
enthalpy() (*burnman.eos.slb.SLB2 method*), 460
enthalpy() (*burnman.eos.slb.SLB3 method*), 465
enthalpy() (*burnman.eos.slb.SLB3Conductive method*), 469
enthalpy() (*burnman.eos.slb.SLBBase method*), 473
enthalpy() (*burnman.eos.spock.SPOCK method*), 395
enthalpy() (*burnman.eos.vinet.Vinet method*), 398

entropy (*burnman.AnisotropicMaterial property*), 302
entropy (*burnman.AnisotropicMineral property*), 320
entropy (*burnman.AnisotropicSolution property*), 516
entropy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
entropy (*burnman.CombinedMineral property*), 264
entropy (*burnman.Composite property*), 337
entropy (*burnman.ElasticSolution property*), 284
entropy (*burnman.Material property*), 233
entropy (*burnman.Mineral property*), 257
entropy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
entropy (*burnman.PerplexMaterial property*), 244
entropy (*burnman.RelaxedAnisotropicSolution property*), 554
entropy (*burnman.RelaxedComposite property*), 345
entropy (*burnman.RelaxedSolution property*), 539
entropy (*burnman.Solution property*), 275
entropy() (*burnman.eos.aa.AA method*), 408
entropy() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
entropy() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 373
entropy() (*burnman.eos.birch_murnaghan.BM3 method*), 361
entropy() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 365
entropy() (*burnman.eos.birch_murnaghan.BM4 method*), 369
entropy() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
entropy() (*burnman.eos.cork.CORK method*), 417
entropy() (*burnman.eos.dks_liquid.DKS_L method*), 421
entropy() (*burnman.eos.dks_solid.DKS_S method*), 425
entropy() (*burnman.eos.EquationOfState method*), 405
entropy() (*burnman.eos.hp.HP98 method*), 428
entropy() (*burnman.eos.hp.HP_TMT method*), 433
entropy() (*burnman.eos.hp.HP_TMTL method*), 436
entropy() (*burnman.eos.IsothermalEquationOfState method*), 355
entropy() (*burnman.eos.macaw.MACAW method*), 377
entropy() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 440
entropy() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 445
entropy() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
entropy() (*burnman.eos.modified_tait.MT method*), 380
entropy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
entropy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
entropy() (*burnman.eos.morse_potential.Morse method*), 384
entropy() (*burnman.eos.murnaghan.Murnaghan method*), 388
entropy() (*burnman.eos.reciprocal_kprime.RKprime method*), 391
entropy() (*burnman.eos.slb.SLB2 method*), 461
entropy() (*burnman.eos.slb.SLB3 method*), 466
entropy() (*burnman.eos.slb.SLB3Conductive method*), 470
entropy() (*burnman.eos.slb.SLBBase method*), 472
entropy() (*burnman.eos.spock.SPOCK method*), 395
entropy() (*burnman.eos.vinet.Vinet method*), 399
entropy() (*in module burnman.eos.bukowinski_electronic*), 480
entropy() (*in module burnman.eos.debye*), 481
entropy() (*in module burnman.eos.einstein*), 482
entropy_hessian (*burnman.AnisotropicSolution property*), 516

entropy_hessian (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 774
entropy_hessian (*burnman.RelaxedAnisotropicSolution property*), 554
entropy_hessian (*burnman.RelaxedSolution property*), 539
entropy_hessian (*burnman.Solution property*), 270
entropy_hessian() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 583
entropy_hessian() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 606
entropy_hessian() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
entropy_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
entropy_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 589
entropy_hessian() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
entropy_hessian() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
entropy_hessian() (*burnman.classes.solutionmodel.IdealSolution method*), 571
entropy_hessian() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
entropy_hessian() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 586
EquationOfState (*class in burnman.eos*), 401
equilibrate() (*in module burnman.tools.equilibration*), 808
equilibrated (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 290
equilibrated (*burnman.Composite property*), 335
equilibrated (*burnman.RelaxedComposite property*), 345
equilibrium_pressure() (*in module burnman.tools.chemistry*), 816
equilibrium_temperature() (*in module burnman.tools.chemistry*), 817
error() (*in module contrib.CHRU2014.paper_fit_data*), 220
estimate_conditions() (*in module burnman.tools.thermobarometry*), 812
evaluate() (*burnman.AnisotropicMaterial method*), 302
evaluate() (*burnman.AnisotropicMineral method*), 320
evaluate() (*burnman.AnisotropicSolution method*), 517
evaluate() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 290
evaluate() (*burnman.classes.seismic.AK135 method*), 683
evaluate() (*burnman.classes.seismic.Fast method*), 673
evaluate() (*burnman.classes.seismic.IASP91 method*), 680
evaluate() (*burnman.classes.seismic.PREM method*), 667
evaluate() (*burnman.classes.seismic.Seismic1DModel method*), 659
evaluate() (*burnman.classes.seismic.SeismicTable method*), 665
evaluate() (*burnman.classes.seismic.Slow method*), 670
evaluate() (*burnman.classes.seismic.STW105 method*), 676
evaluate() (*burnman.CombinedMineral method*), 264
evaluate() (*burnman.Composite method*), 338
evaluate() (*burnman.ElasticSolution method*), 284
evaluate() (*burnman.Layer method*), 631
evaluate() (*burnman.Material method*), 225
evaluate() (*burnman.Mineral method*), 257
evaluate() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 774
evaluate() (*burnman.PerplexMaterial method*),

244
evaluate() (*burnman.Planet method*), 639
evaluate() (*burnman.RelaxedAnisotropicSolution method*), 554
evaluate() (*burnman.RelaxedComposite method*), 345
evaluate() (*burnman.RelaxedSolution method*), 539
evaluate() (*burnman.Solution method*), 275
evaluate_with_volumes() (*burnman.AnisotropicMaterial method*), 303
evaluate_with_volumes() (*burnman.AnisotropicMineral method*), 321
evaluate_with_volumes() (*burnman.AnisotropicSolution method*), 517
evaluate_with_volumes() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 290
evaluate_with_volumes() (*burnman.CombinedMineral method*), 264
evaluate_with_volumes() (*burnman.Composite method*), 338
evaluate_with_volumes() (*burnman.ElasticSolution method*), 284
evaluate_with_volumes() (*burnman.Material method*), 225
evaluate_with_volumes() (*burnman.Mineral method*), 258
evaluate_with_volumes() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 774
evaluate_with_volumes() (*burnman.PerplexMaterial method*), 244
evaluate_with_volumes() (*burnman.RelaxedAnisotropicSolution method*), 554
evaluate_with_volumes() (*burnman.RelaxedComposite method*), 346
evaluate_with_volumes() (*burnman.RelaxedSolution method*), 540
evaluate_with_volumes() (*burnman.Solution method*), 276
examples.example_anisotropic_mineral module, 175
examples.example_anisotropy module, 174
examples.example_averaging module, 183
examples.example_beginner module, 179
examples.example_build_planet module, 194
examples.example_calibrants module, 174
examples.example_chemical_potentials module, 184
examples.example_compare_all_methods module, 193
examples.example_composite module, 172
examples.example_composite_seismic_velocities module, 182
examples.example_composition module, 178
examples.example_equilibrate module, 207
examples.example_fit_composition module, 196
examples.example_fit_data module, 197
examples.example_fit_eos module, 200
examples.example_fit_solution module, 207
examples.example_geotherms module, 176
examples.example_gibbs_modifiers module, 166
examples.example_grid module, 221
examples.example_hyperelastic_stress_rates module, 186
examples.example_mineral module, 165
examples.example_olivine_binary module, 219
examples.example_optimize_pv module, 192
examples.example_seismic module, 180
examples.example_solution module, 169
examples.example_spintransition module, 190
examples.example_spintransition_thermal module, 191

examples.example_user_input_material module, 192
examples.example_woutput module, 221
excess_enthalpy (*burnman.AnisotropicSolution property*), 517
excess_enthalpy (*burnman.ElasticSolution property*), 281
excess_enthalpy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
excess_enthalpy (*burnman.RelaxedAnisotropicSolution property*), 555
excess_enthalpy (*burnman.RelaxedSolution property*), 540
excess_enthalpy (*burnman.Solution property*), 271
excess_enthalpy() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 580
excess_enthalpy() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
excess_enthalpy() (*burnman.classes.solutionmodel.IdealSolution method*), 573
excess_enthalpy() (*burnman.classes.solutionmodel.MechanicalSolution method*), 565
excess_enthalpy() (*burnman.classes.solutionmodel.SubregularSolution method*), 594
excess_enthalpy() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 586
excess_enthalpy() (*burnman.SolutionModel method*), 529
excess_entropy (*burnman.AnisotropicSolution property*), 517
excess_entropy (*burnman.ElasticSolution property*), 281
excess_entropy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
excess_entropy (*burnman.RelaxedAnisotropicSolution property*), 555
excess_entropy (*burnman.RelaxedSolution property*), 540
excess_entropy (*burnman.Solution property*), 271
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegular method*), 583
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 605
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolutio method*), 568
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
excess_entropy() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegular method*), 589
excess_entropy() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 580
excess_entropy() (*burnman.classes.solutionmodel.FunctionSolution method*), 602
excess_entropy() (*burnman.classes.solutionmodel.IdealSolution method*), 573
excess_entropy() (*burnman.classes.solutionmodel.MechanicalSolution method*), 565
excess_entropy() (*burnman.classes.solutionmodel.SubregularSolution method*), 595
excess_entropy() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 586
excess_entropy() (*burnman.ElasticSolutionModel method*), 533
excess_entropy() (*burnman.SolutionModel method*), 529
excess_gibbs (*burnman.AnisotropicSolution property*), 518
excess_gibbs (*burnman.optimize.composition_fitting.DummyCompositionSolut

property), 775
excess_gibbs (*burnman.RelaxedAnisotropicSolution property*), 555
excess_gibbs (*burnman.RelaxedSolution property*), 540
excess_gibbs (*burnman.Solution property*), 270
excess_gibbs_energy() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 580
excess_gibbs_energy() (*burnman.classes.solutionmodel.FunctionSolution method*), 603
excess_gibbs_energy() (*burnman.classes.solutionmodel.IdealSolution method*), 573
excess_gibbs_energy() (*burnman.classes.solutionmodel.MechanicalSolution method*), 564
excess_gibbs_energy() (*burnman.classes.solutionmodel.SubregularSolution method*), 595
excess_gibbs_energy() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 587
excess_gibbs_energy() (*burnman.SolutionModel method*), 528
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 583
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 605
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 576
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolution method*), 568
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
excess_helmholtz_energy() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 589
excess_helmholtz_energy() (*burnman.ElasticSolutionModel method*), 532
excess_partial_entropies (*burnman.AnisotropicSolution property*), 518
excess_partial_entropies (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 775
excess_partial_entropies (*burnman.RelaxedAnisotropicSolution property*), 555
excess_partial_entropies (*burnman.RelaxedSolution property*), 540
excess_partial_entropies (*burnman.Solution property*), 270
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegula method*), 582
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 605
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolutio method*), 569
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 597
excess_partial_entropies() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularS method*), 590
excess_partial_entropies() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 578
excess_partial_entropies() (*burnman.classes.solutionmodel.FunctionSolution method*), 601
excess_partial_entropies() (*burnman.classes.solutionmodel.IdealSolution method*), 571
excess_partial_entropies() (*burnman.classes.solutionmodel.MechanicalSolution method*), 566
excess_partial_entropies() (*burnman.classes.solutionmodel.SubregularSolution method*), 592
excess_partial_entropies() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 587

excess_partial_entropies() (*burnman.ElasticSolutionModel method*), 533
excess_partial_entropies() (*burnman.SolutionModel method*), 530
excess_partial_gibbs (*burnman.AnisotropicSolution property*), 518
excess_partial_gibbs (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
excess_partial_gibbs (*burnman.RelaxedAnisotropicSolution property*), 555
excess_partial_gibbs (*burnman.RelaxedSolution property*), 540
excess_partial_gibbs (*burnman.Solution property*), 270
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 577
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.FunctionSolution method*), 601
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.IdealSolution method*), 570
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.MechanicalSolution method*), 565
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.SubregularSolution method*), 592
excess_partial_gibbs_energies() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 587
excess_partial_gibbs_energies() (*burnman.SolutionModel method*), 530
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 582
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 604
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 574
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolution method*), 569
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 597
excess_partial_helmholtz_energies() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 590
excess_partial_helmholtz_energies() (*burnman.ElasticSolutionModel method*), 533
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 582
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 604
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolution method*), 569
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
excess_partial_pressures() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 590
excess_partial_pressures() (*burnman.ElasticSolutionModel method*), 534
excess_partial_volumes (*burnman.AnisotropicSolution property*), 518
excess_partial_volumes (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
excess_partial_volumes (*burnman.RelaxedAnisotropicSolution property*), 555
excess_partial_volumes (*burnman.RelaxedSolution property*), 540
excess_partial_volumes (*burnman.Solution property*), 270
excess_partial_volumes() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 578
excess_partial_volumes() (*burnman.classes.solutionmodel.FunctionSolution*

method), 600
excess_partial_volumes() (*burnman.classes.solutionmodel.IdealSolution method*), 571
excess_partial_volumes() (*burnman.classes.solutionmodel.MechanicalSolution method*), 566
excess_partial_volumes() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
excess_partial_volumes() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 588
excess_partial_volumes() (*burnman.SolutionModel method*), 531
excess_pressure (*burnman.ElasticSolution property*), 280
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 584
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 606
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 576
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticMechanicalSolution method*), 568
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 599
excess_pressure() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 591
excess_pressure() (*burnman.ElasticSolutionModel method*), 532
excess_volume (*burnman.AnisotropicSolution property*), 518
excess_volume (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
excess_volume (*burnman.RelaxedAnisotropicSolution property*), 555
excess_volume (*burnman.RelaxedSolution property*), 540
excess_volume (*burnman.Solution property*), 271
excess_volume() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 581
excess_volume() (*burnman.classes.solutionmodel.FunctionSolution method*), 603
excess_volume() (*burnman.classes.solutionmodel.IdealSolution method*), 574
excess_volume() (*burnman.classes.solutionmodel.MechanicalSolution method*), 564
excess_volume() (*burnman.classes.solutionmodel.SubregularSolution method*), 595
excess_volume() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 588
excess_volume() (*burnman.SolutionModel method*), 529
expand_strains() (*in module burnman.utils.anisotropy*), 833
expand_stresses() (*in module burnman.utils.anisotropy*), 833
extract_lines_between_markers() (*in module burnman.utils.misc*), 844
extreme_values() (*in module burnman.optimize.nonlinear_fitting*), 797
eye (*burnman.AnisotropicSolution property*), 512
eye (*burnman.classes.solutionmodel.AsymmetricRegularSolution property*), 581
eye (*burnman.classes.solutionmodel.FunctionSolution property*), 603
eye (*burnman.classes.solutionmodel.IdealSolution property*), 572
eye (*burnman.classes.solutionmodel.SubregularSolution property*), 596
eye (*burnman.classes.solutionmodel.SymmetricRegularSolution property*), 588
eye (*burnman.RelaxedAnisotropicSolution property*), 555
eyeones (*burnman.classes.solutionmodel.AsymmetricRegularSolution property*), 581
eyeones (*burnman.classes.solutionmodel.FunctionSolution property*), 603
eyeones (*burnman.classes.solutionmodel.IdealSolution property*), 572

eyeones (*burnman.classes.solutionmodel.SubregularSolution property*), 596
eyeones (*burnman.classes.solutionmodel.SymmetricRegularSolution property*), 588

F

F (*burnman.optimize.nonlinear_solvers.Iterates attribute*), 798
F (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
F() (*in module burnman.tools.equilibration*), 803
F_norm (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
Fast (*class in burnman.classes.seismic*), 672
file_to_composition_list() (*in module burnman.classes.composition*), 609
find_mle() (*in module burnman.optimize.nonlinear_fitting*), 792
fit_composition_to_solution() (*in module burnman.optimize.composition_fitting*), 781
fit_phase_proportions_to_bulk_composition() (*in module burnman.optimize.composition_fitting*), 782
fit_PTp_data() (*in module burnman.optimize.eos_fitting*), 784
fit_PTV_data() (*in module burnman.optimize.eos_fitting*), 785
fit_VTP_data() (*in module burnman.optimize.eos_fitting*), 788
fit_VTp_data() (*in module burnman.optimize.eos_fitting*), 787
fit_XPTp_data() (*in module burnman.optimize.eos_fitting*), 789
flatten() (*in module burnman.utils.misc*), 842
formula (*burnman.AnisotropicMineral property*), 321
formula (*burnman.AnisotropicSolution property*), 518
formula (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
formula (*burnman.CombinedMineral property*), 261
formula (*burnman.Composite property*), 333
formula (*burnman.ElasticSolution property*), 280
formula (*burnman.Mineral property*), 252
formula (*burnman.optimize.composition_fitting.DummyComposition property*), 775
formula (*burnman.RelaxedAnisotropicSolution property*), 555
formula (*burnman.RelaxedComposite property*), 346
formula (*burnman.RelaxedSolution property*), 540
formula (*burnman.Solution property*), 269
formula_mass() (*in module burnman.utils.chemistry*), 829
formula_to_string() (*in module burnman.utils.chemistry*), 831
fromkeys() (*burnman.utils.misc.OrderedCounter class method*), 841
fugacity() (*in module burnman.tools.chemistry*), 816
full_isentropic_compliance_tensor (*burnman.AnisotropicMaterial property*), 297
full_isentropic_compliance_tensor (*burnman.AnisotropicMineral property*), 316
full_isentropic_compliance_tensor (*burnman.AnisotropicSolution property*), 518
full_isentropic_compliance_tensor (*burnman.RelaxedAnisotropicSolution property*), 555
full_isentropic_stiffness_tensor (*burnman.AnisotropicMaterial property*), 297
full_isentropic_stiffness_tensor (*burnman.AnisotropicMineral property*), 317
full_isentropic_stiffness_tensor (*burnman.AnisotropicSolution property*), 518
full_isentropic_stiffness_tensor (*burnman.RelaxedAnisotropicSolution property*), 555
full_isothermal_compliance_tensor (*burnman.AnisotropicMineral property*), 316
full_isothermal_compliance_tensor (*burnman.AnisotropicSolution property*), 518
full_isothermal_compliance_tensor (*burnman.RelaxedAnisotropicSolution property*), 547
full_isothermal_stiffness_tensor (*burnman.AnisotropicMineral property*), 316
full_isothermal_stiffness_tensor (*burnman.AnisotropicSolution property*), 518
full_isothermal_stiffness_tensor (*burnman.RelaxedAnisotropicSolution property*), 555
fvalue_vp() (*burnman.optimize.eos_fitting.MineralFit*

method), 784
function() (*burnman.optimize.eos_fitting.MineralFitV method*), 786
function() (*burnman.optimize.eos_fitting.SolutionFit method*), 789
function() (*burnman.optimize.nonlinear_fitting.NonLinearModel method*), 791
FunctionSolution (*class in burnman.classes.solutionmodel*), 599

G

G (*burnman.AnisotropicMaterial property*), 300
G (*burnman.AnisotropicMineral property*), 318
G (*burnman.AnisotropicSolution property*), 512
G (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
G (*burnman.CombinedMineral property*), 262
G (*burnman.Composite property*), 337
G (*burnman.ElasticSolution property*), 282
G (*burnman.Layer property*), 637
G (*burnman.Material property*), 234
G (*burnman.Mineral property*), 256
G (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
G (*burnman.PerplexMaterial property*), 242
G (*burnman.Planet property*), 646
G (*burnman.RelaxedAnisotropicSolution property*), 549
G (*burnman.RelaxedComposite property*), 343
G (*burnman.RelaxedSolution property*), 536
G (*burnman.Solution property*), 273
G() (*burnman.classes.seismic.AK135 method*), 681
G() (*burnman.classes.seismic.Fast method*), 672
G() (*burnman.classes.seismic.IASP91 method*), 678
G() (*burnman.classes.seismic.PREM method*), 665
G() (*burnman.classes.seismic.Seismic1DModel method*), 661
G() (*burnman.classes.seismic.SeismicTable method*), 664
G() (*burnman.classes.seismic.Slow method*), 669
G() (*burnman.classes.seismic.STW105 method*), 675
G_eff (*burnman.AnisotropicMaterial property*), 300
G_eff (*burnman.AnisotropicMineral property*), 318
G_eff (*burnman.AnisotropicSolution property*), 512
G_eff (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
G_eff (*burnman.CombinedMineral property*), 262
G_eff (*burnman.Composite property*), 337
G_eff (*burnman.ElasticSolution property*), 282
G_eff (*burnman.Material property*), 234
G_eff (*burnman.Mineral property*), 256
G_eff (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
G_eff (*burnman.PerplexMaterial property*), 242
G_eff (*burnman.RelaxedAnisotropicSolution property*), 549
G_eff (*burnman.RelaxedComposite property*), 343
G_eff (*burnman.RelaxedSolution property*), 536
G_eff (*burnman.Solution property*), 274
generalised_gammainc() (*in module burnman.utils.math*), 840
generate_complete_basis() (*in module burnman.utils.math*), 839
Geotherm (*class in burnman.classes.geotherm*), 623
GeothermFromPressures (*class in burnman.classes.geotherm*), 626
get() (*burnman.utils.misc.OrderedCounter method*), 841
get_endmember_amounts() (*in module burnman.tools.equilibration*), 803
get_equilibration_parameters() (*in module burnman.tools.equilibration*), 806
get_layer() (*burnman.Planet method*), 639
get_layer_by_radius() (*burnman.Planet method*), 639
get_parameters() (*in module burnman.tools.equilibration*), 803
get_params() (*burnman.optimize.eos_fitting.MineralFit method*), 784
get_params() (*burnman.optimize.eos_fitting.MineralFitV method*), 786
get_params() (*burnman.optimize.eos_fitting.SolutionFit method*), 788
get_params() (*burnman.optimize.nonlinear_fitting.NonLinearModel method*), 791
get_reaction_matrix() (*in module burnman.tools.thermobarometry*), 809

gibbs (*burnman.AnisotropicMaterial property*), 303
gibbs (*burnman.AnisotropicMineral property*), 321
gibbs (*burnman.AnisotropicSolution property*), 518
gibbs (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
gibbs (*burnman.CombinedMineral property*), 264
gibbs (*burnman.Composite property*), 338
gibbs (*burnman.ElasticSolution property*), 285
gibbs (*burnman.Layer property*), 636
gibbs (*burnman.Material property*), 233
gibbs (*burnman.Mineral property*), 258
gibbs (*burnman.optimize.composition_fitting.DummyCombinedSolution property*), 775
gibbs (*burnman.PerplexMaterial property*), 245
gibbs (*burnman.Planet property*), 645
gibbs (*burnman.RelaxedAnisotropicSolution property*), 555
gibbs (*burnman.RelaxedComposite property*), 346
gibbs (*burnman.RelaxedSolution property*), 540
gibbs (*burnman.Solution property*), 276
gibbs_energy() (*burnman.eos.aa.AA method*), 409
gibbs_energy() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
gibbs_energy() (*burnman.eos.birch_murnaghan.BM3 method*), 361
gibbs_energy() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 365
gibbs_energy() (*burnman.eos.birch_murnaghan.BM4 method*), 370
gibbs_energy() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
gibbs_energy() (*burnman.eos.cork.CORK method*), 416
gibbs_energy() (*burnman.eos.dks_liquid.DKS_L method*), 421
gibbs_energy() (*burnman.eos.dks_solid.DKS_S method*), 425
gibbs_energy() (*burnman.eos.EquationOfState method*), 404
gibbs_energy() (*burnman.eos.hp.HP98 method*), 428
gibbs_energy() (*burnman.eos.hp.HP_TMT method*), 433
gibbs_energy() (*burnman.eos.hp.HP_TMTL method*), 436
gibbs_energy() (*burnman.eos.IsothermalEquationOfState method*), 356
gibbs_energy() (*burnman.eos.macaw.MACAW method*), 376
gibbs_energy() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 440
gibbs_energy() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 445
gibbs_energy() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
gibbs_energy() (*burnman.eos.modified_tait.MT method*), 379
gibbs_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
gibbs_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
gibbs_energy() (*burnman.eos.morse_potential.Morse method*), 383
gibbs_energy() (*burnman.eos.murnaghan.Murnaghan method*), 387
gibbs_energy() (*burnman.eos.reciprocal_kprime.RKprime method*), 391
gibbs_energy() (*burnman.eos.slb.SLB2 method*), 461
gibbs_energy() (*burnman.eos.slb.SLB3 method*), 466
gibbs_energy() (*burnman.eos.slb.SLB3Conductive method*), 470
gibbs_energy() (*burnman.eos.slb.SLBBase method*), 472
gibbs_energy() (*burnman.eos.spock.SPOCK method*), 394
gibbs_energy() (*burnman.eos.vinet.Vinet method*), 398
gibbs_hessian (*burnman.AnisotropicSolution*

property), 519
gibbs_hessian (burnman.ElasticSolution property), 280
gibbs_hessian (burnman.optimize.composition_fitting.DummyCompositionSolution property), 775
gibbs_hessian (burnman.RelaxedAnisotropicSolution property), 556
gibbs_hessian (burnman.RelaxedSolution property), 540
gibbs_hessian (burnman.Solution property), 270
gibbs_hessian() (burnman.classes.solutionmodel.AsymmetricRegularSolution method), 578
gibbs_hessian() (burnman.classes.solutionmodel.FunctionSolution method), 602
gibbs_hessian() (burnman.classes.solutionmodel.IdealSolution method), 571
gibbs_hessian() (burnman.classes.solutionmodel.SubregularSolution method), 593
gibbs_hessian() (burnman.classes.solutionmodel.SymmetricRegularSolution method), 589
gr (burnman.AnisotropicMaterial property), 303
gr (burnman.AnisotropicMineral property), 321
gr (burnman.AnisotropicSolution property), 519
gr (burnman.classes.mineral_helpers.HelperSpinTransition property), 291
gr (burnman.CombinedMineral property), 265
gr (burnman.Composite property), 338
gr (burnman.ElasticSolution property), 285
gr (burnman.Layer property), 637
gr (burnman.Material property), 234
gr (burnman.Mineral property), 258
gr (burnman.optimize.composition_fitting.DummyCompositionSolution property), 775
gr (burnman.PerplexMaterial property), 245
gr (burnman.Planet property), 646
gr (burnman.RelaxedAnisotropicSolution property), 556
gr (burnman.RelaxedComposite property), 346
gr (burnman.RelaxedSolution property), 541
gr (burnman.Solution property), 276
gravity (burnman.Layer property), 631
gravity (burnman.Planet property), 640
gravity() (burnman.classes.seismic.AK135 method), 683
gravity() (burnman.classes.seismic.Fast method), 674
gravity() (burnman.classes.seismic.IASP91 method), 680
gravity() (burnman.classes.seismic.PREM method), 667
gravity() (burnman.classes.seismic.Seismic1DModel method), 662
gravity() (burnman.classes.seismic.SeismicTable method), 663
gravity() (burnman.classes.seismic.Slow method), 670
gravity() (burnman.classes.seismic.STW105 method), 677
greedy_independent_endmember_selection() (in module burnman.tools.polytope), 825
grid() (burnman.MaterialPolytope method), 613
grueneisen_parameter (burnman.AnisotropicMaterial property), 303
grueneisen_parameter (burnman.AnisotropicMineral property), 317
grueneisen_parameter (burnman.AnisotropicSolution property), 519
grueneisen_parameter (burnman.classes.mineral_helpers.HelperSpinTransition property), 291
grueneisen_parameter (burnman.CombinedMineral property), 262
grueneisen_parameter (burnman.Composite property), 334
grueneisen_parameter (burnman.ElasticSolution property), 281
grueneisen_parameter (burnman.Layer property), 636
grueneisen_parameter (burnman.Material property), 232
grueneisen_parameter (burnman.Mineral property), 255
grueneisen_parameter (burnman.optimize.composition_fitting.DummyCompositionSolution property), 775
grueneisen_parameter (burnman.PerplexMaterial property), 242

grueneisen_parameter (*burnman.Planet property*), 644
grueneisen_parameter (*burnman.RelaxedAnisotropicSolution property*), 556
grueneisen_parameter (*burnman.RelaxedComposite property*), 347
grueneisen_parameter (*burnman.RelaxedSolution property*), 541
grueneisen_parameter (*burnman.Solution property*), 271
grueneisen_parameter() (*burnman.eos.aa.AA method*), 409
grueneisen_parameter() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 373
grueneisen_parameter() (*burnman.eos.birch_murnaghan.BM3 method*), 361
grueneisen_parameter() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 365
grueneisen_parameter() (*burnman.eos.birch_murnaghan.BM4 method*), 370
grueneisen_parameter() (*burnman.eos.brosh_calphad.BroshCalphad method*), 414
grueneisen_parameter() (*burnman.eos.cork.CORK method*), 417
grueneisen_parameter() (*burnman.eos.dks_liquid.DKS_L method*), 422
grueneisen_parameter() (*burnman.eos.dks_solid.DKS_S method*), 425
grueneisen_parameter() (*burnman.eos.EquationOfState method*), 402
grueneisen_parameter() (*burnman.eos.hp.HP98 method*), 429
grueneisen_parameter() (*burnman.eos.hp.HP_TMT method*), 433
grueneisen_parameter() (*burnman.eos.hp.HP_TMTL method*), 437
grueneisen_parameter() (*burnman.eos.IsothermalEquationOfState method*), 356
grueneisen_parameter() (*burnman.eos.macaw.MACAW method*), 377
grueneisen_parameter() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 440
grueneisen_parameter() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 445
grueneisen_parameter() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 448
grueneisen_parameter() (*burnman.eos.modified_tait.MT method*), 380
grueneisen_parameter() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
grueneisen_parameter() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
grueneisen_parameter() (*burnman.eos.morse_potential.Morse method*), 384
grueneisen_parameter() (*burnman.eos.murnaghan.Murnaghan method*), 388
grueneisen_parameter() (*burnman.eos.reciprocal_kprime.RKprime method*), 391
grueneisen_parameter() (*burnman.eos.slb.SLB2 method*), 461
grueneisen_parameter() (*burnman.eos.slb.SLB3 method*), 466
grueneisen_parameter() (*burnman.eos.slb.SLB3Conductive method*), 470
grueneisen_parameter() (*burnman.eos.slb.SLBBase method*), 473
grueneisen_parameter() (*burnman.eos.spock.SPOCK method*), 395
grueneisen_parameter() (*burnman.eos.vinet.Vinet method*), 399
grueneisen_tensor (*burnman.AnisotropicMineral property*), 317
grueneisen_tensor (*burnman.AnisotropicSolution property*), 519
grueneisen_tensor (*burnman.RelaxedAnisotropicSolution property*), 556

H

H (*burnman.AnisotropicMaterial property*), 300
H (*burnman.AnisotropicMineral property*), 318
H (*burnman.AnisotropicSolution property*), 512
H (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
H (*burnman.CombinedMineral property*), 262
H (*burnman.Composite property*), 337
H (*burnman.ElasticSolution property*), 283
H (*burnman.Layer property*), 637
H (*burnman.Material property*), 234
H (*burnman.Mineral property*), 256
H (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
H (*burnman.PerplexMaterial property*), 242
H (*burnman.Planet property*), 645
H (*burnman.RelaxedAnisotropicSolution property*), 549
H (*burnman.RelaxedComposite property*), 343
H (*burnman.RelaxedSolution property*), 536
H (*burnman.Solution property*), 274
HashinShtrikmanAverage (*class in burnman.averaging_schemes*), 622
HashinShtrikmanLower (*class in burnman.averaging_schemes*), 621
HashinShtrikmanUpper (*class in burnman.averaging_schemes*), 620
heat_capacity_p (*burnman.AnisotropicMaterial property*), 303
heat_capacity_p (*burnman.AnisotropicMineral property*), 321
heat_capacity_p (*burnman.AnisotropicSolution property*), 519
heat_capacity_p (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
heat_capacity_p (*burnman.CombinedMineral property*), 265
heat_capacity_p (*burnman.Composite property*), 339
heat_capacity_p (*burnman.ElasticSolution property*), 285
heat_capacity_p (*burnman.Material property*), 233
heat_capacity_p (*burnman.Mineral property*), 258
heat_capacity_p (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 775
heat_capacity_p (*burnman.PerplexMaterial property*), 245
heat_capacity_p (*burnman.RelaxedAnisotropicSolution property*), 556
heat_capacity_p (*burnman.RelaxedComposite property*), 347
heat_capacity_p (*burnman.RelaxedSolution property*), 541
heat_capacity_p (*burnman.Solution property*), 276
heat_capacity_v (*burnman.AnisotropicMaterial property*), 303
heat_capacity_v (*burnman.AnisotropicMineral property*), 321
heat_capacity_v (*burnman.AnisotropicSolution property*), 519
heat_capacity_v (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
heat_capacity_v (*burnman.CombinedMineral property*), 265
heat_capacity_v (*burnman.Composite property*), 339
heat_capacity_v (*burnman.ElasticSolution property*), 285
heat_capacity_v (*burnman.Material property*), 233
heat_capacity_v (*burnman.Mineral property*), 258
heat_capacity_v (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
heat_capacity_v (*burnman.PerplexMaterial property*), 245
heat_capacity_v (*burnman.RelaxedAnisotropicSolution property*), 556
heat_capacity_v (*burnman.RelaxedComposite property*), 347
heat_capacity_v (*burnman.RelaxedSolution property*), 541
heat_capacity_v (*burnman.Solution property*), 276
heat_capacity_v() (*burnman.eos.anharmonic_debye.AnharmonicDebye*

static method), 476
helmholtz (*burnman.AnisotropicMaterial property*), 303
helmholtz (*burnman.AnisotropicMineral property*), 321
helmholtz (*burnman.AnisotropicSolution property*), 519
helmholtz (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
helmholtz (*burnman.CombinedMineral property*), 265
helmholtz (*burnman.Composite property*), 339
helmholtz (*burnman.ElasticSolution property*), 285
helmholtz (*burnman.Layer property*), 636
helmholtz (*burnman.Material property*), 233
helmholtz (*burnman.Mineral property*), 258
helmholtz (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
helmholtz (*burnman.PerplexMaterial property*), 245
helmholtz (*burnman.Planet property*), 645
helmholtz (*burnman.RelaxedAnisotropicSolution property*), 556
helmholtz (*burnman.RelaxedComposite property*), 347
helmholtz (*burnman.RelaxedSolution property*), 541
helmholtz (*burnman.Solution property*), 276
helmholtz() (*in module burnman.eos.bukowinski_electronic*), 480
helmholtz_energy() (*burnman.eos.aa.AA method*), 410
helmholtz_energy() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
helmholtz_energy() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 373
helmholtz_energy() (*burnman.eos.birch_murnaghan.BM3 method*), 361
helmholtz_energy() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 366
helmholtz_energy() (*burnman.eos.birch_murnaghan.BM4 method*), 370
helmholtz_energy() (*burnman.eos.brosh_calphad.BroshCalphad method*), 414
helmholtz_energy() (*burnman.eos.cork.CORK method*), 418
helmholtz_energy() (*burnman.eos.dks_liquid.DKS_L method*), 422
helmholtz_energy() (*burnman.eos.dks_solid.DKS_S method*), 426
helmholtz_energy() (*burnman.eos.EquationOfState method*), 405
helmholtz_energy() (*burnman.eos.hp.HP98 method*), 429
helmholtz_energy() (*burnman.eos.hp.HP_TMT method*), 434
helmholtz_energy() (*burnman.eos.hp.HP_TMTL method*), 437
helmholtz_energy() (*burnman.eos.IsothermalEquationOfState method*), 356
helmholtz_energy() (*burnman.eos.macaw.MACAW method*), 377
helmholtz_energy() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 440
helmholtz_energy() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 445
helmholtz_energy() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 449
helmholtz_energy() (*burnman.eos.modified_tait.MT method*), 381
helmholtz_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 453
helmholtz_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 458
helmholtz_energy() (*burnman.eos.morse_potential.Morse method*), 384
helmholtz_energy() (*burn-*

man.eos.murnaghan.Murnaghan method), 388
helmholtz_energy() (*burnman.eos.reciprocal_kprime.RKprime method*), 392
helmholtz_energy() (*burnman.eos.slb.SLB2 method*), 461
helmholtz_energy() (*burnman.eos.slb.SLB3 method*), 466
helmholtz_energy() (*burnman.eos.slb.SLB3Conductive method*), 470
helmholtz_energy() (*burnman.eos.slb.SLBBase method*), 474
helmholtz_energy() (*burnman.eos.spock.SPOCK method*), 395
helmholtz_energy() (*burnman.eos.vinet.Vinet method*), 399
helmholtz_energy() (*in module burnman.eos.debye*), 481
helmholtz_energy() (*in module burnman.eos.einstein*), 482
helmholtz_hessian() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegularSolution method*), 583
helmholtz_hessian() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 606
helmholtz_hessian() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
helmholtz_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
helmholtz_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 591
HelperSpinTransition (*class in burnman.classes.mineral_helpers*), 288
HP98 (*class in burnman.eos.hp*), 427
HP_TMT (*class in burnman.eos.hp*), 431
HP_TMTL (*class in burnman.eos.hp*), 435
hugoniot() (*in module burnman.tools.chemistry*), 818

I

IASP91 (*class in burnman.classes.seismic*), 678
IdealSolution (*class in burnman.classes.solutionmodel*), 570
independent_element_indices (*burnman.AnisotropicSolution property*), 519
independent_element_indices (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 291
independent_element_indices (*burnman.Composite property*), 336
independent_element_indices (*burnman.ElasticSolution property*), 282
independent_element_indices (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 776
independent_element_indices (*burnman.RelaxedAnisotropicSolution property*), 556
independent_element_indices (*burnman.RelaxedComposite property*), 347
independent_element_indices (*burnman.RelaxedSolution property*), 541
independent_element_indices (*burnman.Solution property*), 273
independent_endmember_limits (*burnman.MaterialPolytope property*), 613
independent_endmember_occupancies (*burnman.MaterialPolytope property*), 613
independent_endmember_polytope (*burnman.MaterialPolytope property*), 613
independent_row_indices() (*in module burnman.utils.math*), 839
independent_row_indices() (*in module burnman.utils.reductions*), 840
integrate_rates() (*in module examples.example_hyperelastic_stress_rates*), 189
internal_depth_list() (*burnman.classes.seismic.AK135 method*), 683
internal_depth_list() (*burnman.classes.seismic.Fast method*), 674
internal_depth_list() (*burnman.classes.seismic.IASP91 method*), 680
internal_depth_list() (*burnman.classes.seismic.PREM method*), 667
internal_depth_list() (*burn-*

man.classes.seismic.Seismic1DModel method), 659
internal_depth_list() (*burnman.classes.seismic.SeismicTable method*), 662
internal_depth_list() (*burnman.classes.seismic.Slow method*), 671
internal_depth_list() (*burnman.classes.seismic.STW105 method*), 677
internal_energy (*burnman.AnisotropicMaterial property*), 304
internal_energy (*burnman.AnisotropicMineral property*), 321
internal_energy (*burnman.AnisotropicSolution property*), 519
internal_energy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
internal_energy (*burnman.CombinedMineral property*), 265
internal_energy (*burnman.Composite property*), 339
internal_energy (*burnman.ElasticSolution property*), 285
internal_energy (*burnman.Material property*), 233
internal_energy (*burnman.Mineral property*), 258
internal_energy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
internal_energy (*burnman.PerplexMaterial property*), 245
internal_energy (*burnman.RelaxedAnisotropicSolution property*), 556
internal_energy (*burnman.RelaxedComposite property*), 347
internal_energy (*burnman.RelaxedSolution property*), 541
internal_energy (*burnman.Solution property*), 276
interp_smoothed_array_and_derivatives() (*in module burnman.utils.math*), 836
intVdP() (*in module burnman.eos.modified_tait*), 482
intVdP() (*in module burnman.eos.murnaghan*), 483
invariant_point() (*in module burnman.tools.chemistry*), 817
is_positive_definite() (*in module burnman.utils.math*), 840
isentropic_bulk_modulus_reuss (*burnman.AnisotropicMaterial property*), 298
isentropic_bulk_modulus_reuss (*burnman.AnisotropicMineral property*), 314
isentropic_bulk_modulus_reuss (*burnman.AnisotropicSolution property*), 519
isentropic_bulk_modulus_reuss (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
isentropic_bulk_modulus_reuss (*burnman.CombinedMineral property*), 261
isentropic_bulk_modulus_reuss (*burnman.Composite property*), 334
isentropic_bulk_modulus_reuss (*burnman.ElasticSolution property*), 281
isentropic_bulk_modulus_reuss (*burnman.Layer property*), 634
isentropic_bulk_modulus_reuss (*burnman.Material property*), 229
isentropic_bulk_modulus_reuss (*burnman.Mineral property*), 253
isentropic_bulk_modulus_reuss (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 776
isentropic_bulk_modulus_reuss (*burnman.PerplexMaterial property*), 237
isentropic_bulk_modulus_reuss (*burnman.Planet property*), 643
isentropic_bulk_modulus_reuss (*burnman.RelaxedAnisotropicSolution property*), 548
isentropic_bulk_modulus_reuss (*burnman.RelaxedComposite property*), 347
isentropic_bulk_modulus_reuss (*burnman.RelaxedSolution property*), 536
isentropic_bulk_modulus_reuss (*burnman.Solution property*), 271
isentropic_bulk_modulus_reuss() (*burnman.eos.aa.AA method*), 410
isentropic_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 374
isentropic_bulk_modulus_reuss() (*burn-*

man.eos.birch_murnaghan.BM3 method), 362
isentropic_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 366
isentropic_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BM4 method*), 370
isentropic_bulk_modulus_reuss() (*burnman.eos.brosh_calphad.BroshCalphad method*), 414
isentropic_bulk_modulus_reuss() (*burnman.eos.cork.CORK method*), 418
isentropic_bulk_modulus_reuss() (*burnman.eos.dks_liquid.DKS_L method*), 423
isentropic_bulk_modulus_reuss() (*burnman.eos.dks_solid.DKS_S method*), 426
isentropic_bulk_modulus_reuss() (*burnman.eos.EquationOfState method*), 403
isentropic_bulk_modulus_reuss() (*burnman.eos.hp.HP98 method*), 430
isentropic_bulk_modulus_reuss() (*burnman.eos.hp.HP_TMT method*), 434
isentropic_bulk_modulus_reuss() (*burnman.eos.hp.HP_TMTL method*), 438
isentropic_bulk_modulus_reuss() (*burnman.eos.IsothermalEquationOfState method*), 357
isentropic_bulk_modulus_reuss() (*burnman.eos.macaw.MACAW method*), 377
isentropic_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 441
isentropic_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 445
isentropic_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 449
isentropic_bulk_modulus_reuss() (*burnman.eos.modified_tait.MT method*), 381
isentropic_bulk_modulus_reuss() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 453
isentropic_bulk_modulus_reuss() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDW method*), 458
isentropic_bulk_modulus_reuss() (*burnman.eos.morse_potential.Morse method*), 385
isentropic_bulk_modulus_reuss() (*burnman.eos.murnaghan.Murnaghan method*), 389
isentropic_bulk_modulus_reuss() (*burnman.eos.reciprocal_kprime.RKprime method*), 392
isentropic_bulk_modulus_reuss() (*burnman.eos.slb.SLB2 method*), 461
isentropic_bulk_modulus_reuss() (*burnman.eos.slb.SLB3 method*), 466
isentropic_bulk_modulus_reuss() (*burnman.eos.slb.SLB3Conductive method*), 470
isentropic_bulk_modulus_reuss() (*burnman.eos.slb.SLBBase method*), 474
isentropic_bulk_modulus_reuss() (*burnman.eos.spock.SPOCK method*), 396
isentropic_bulk_modulus_reuss() (*burnman.eos.vinet.Vinet method*), 399
isentropic_bulk_modulus_voigt (*burnman.AnisotropicMaterial property*), 298
isentropic_bulk_modulus_voigt (*burnman.AnisotropicMineral property*), 315
isentropic_bulk_modulus_voigt (*burnman.AnisotropicSolution property*), 519
isentropic_bulk_modulus_voigt (*burnman.RelaxedAnisotropicSolution property*), 556
isentropic_bulk_modulus_vrh (*burnman.AnisotropicMaterial property*), 298
isentropic_bulk_modulus_vrh (*burnman.AnisotropicMineral property*), 321
isentropic_bulk_modulus_vrh (*burnman.AnisotropicSolution property*), 519
isentropic_bulk_modulus_vrh (*burnman.RelaxedAnisotropicSolution property*), 556
isentropic_compliance_tensor (*burnman.AnisotropicMaterial property*), 297
isentropic_compliance_tensor (*burnman.AnisotropicMineral property*), 316
isentropic_compliance_tensor (*burnman.AnisotropicSolution property*), 520

isentropic_compliance_tensor (*burnman.RelaxedAnisotropicSolution property*), 556
isentropic_compressibility_reuss (*burnman.AnisotropicMaterial property*), 304
isentropic_compressibility_reuss (*burnman.AnisotropicMineral property*), 315
isentropic_compressibility_reuss (*burnman.AnisotropicSolution property*), 520
isentropic_compressibility_reuss (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
isentropic_compressibility_reuss (*burnman.CombinedMineral property*), 262
isentropic_compressibility_reuss (*burnman.Composite property*), 334
isentropic_compressibility_reuss (*burnman.ElasticSolution property*), 281
isentropic_compressibility_reuss (*burnman.Layer property*), 635
isentropic_compressibility_reuss (*burnman.Material property*), 230
isentropic_compressibility_reuss (*burnman.Mineral property*), 254
isentropic_compressibility_reuss (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
isentropic_compressibility_reuss (*burnman.PerplexMaterial property*), 241
isentropic_compressibility_reuss (*burnman.Planet property*), 643
isentropic_compressibility_reuss (*burnman.RelaxedAnisotropicSolution property*), 548
isentropic_compressibility_reuss (*burnman.RelaxedComposite property*), 347
isentropic_compressibility_reuss (*burnman.RelaxedSolution property*), 536
isentropic_compressibility_reuss (*burnman.Solution property*), 271
isentropic_compressibility_tensor (*burnman.AnisotropicMineral property*), 317
isentropic_compressibility_tensor (*burnman.AnisotropicSolution property*), 520
isentropic_compressibility_tensor (*burnman.RelaxedAnisotropicSolution property*), 557
isentropic_compressibility_voigt (*burnman.AnisotropicMineral property*), 315
isentropic_compressibility_voigt (*burnman.AnisotropicSolution property*), 520
isentropic_compressibility_voigt (*burnman.RelaxedAnisotropicSolution property*), 557
isentropic_isotropic_poisson_ratio (*burnman.AnisotropicMaterial property*), 299
isentropic_isotropic_poisson_ratio (*burnman.AnisotropicMineral property*), 322
isentropic_isotropic_poisson_ratio (*burnman.AnisotropicSolution property*), 520
isentropic_isotropic_poisson_ratio (*burnman.RelaxedAnisotropicSolution property*), 557
isentropic_linear_compressibility() (*burnman.AnisotropicMaterial method*), 299
isentropic_linear_compressibility() (*burnman.AnisotropicMineral method*), 322
isentropic_linear_compressibility() (*burnman.AnisotropicSolution method*), 520
isentropic_linear_compressibility() (*burnman.RelaxedAnisotropicSolution method*), 557
isentropic_poissons_ratio() (*burnman.AnisotropicMaterial method*), 300
isentropic_poissons_ratio() (*burnman.AnisotropicMineral method*), 322
isentropic_poissons_ratio() (*burnman.AnisotropicSolution method*), 520
isentropic_poissons_ratio() (*burnman.RelaxedAnisotropicSolution method*), 557
isentropic_shear_modulus() (*burnman.AnisotropicMaterial method*), 300
isentropic_shear_modulus() (*burnman.AnisotropicMineral method*), 322
isentropic_shear_modulus() (*burnman.AnisotropicSolution method*), 521
isentropic_shear_modulus() (*burnman.RelaxedAnisotropicSolution method*), 557
isentropic_shear_modulus_reuss (*burnman.AnisotropicMaterial property*), 298
isentropic_shear_modulus_reuss (*burn-*

man.AnisotropicMineral property), 322
isentropic_shear_modulus_reuss (*burnman.AnisotropicSolution property*), 521
isentropic_shear_modulus_reuss (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_shear_modulus_voigt (*burnman.AnisotropicMaterial property*), 298
isentropic_shear_modulus_voigt (*burnman.AnisotropicMineral property*), 322
isentropic_shear_modulus_voigt (*burnman.AnisotropicSolution property*), 521
isentropic_shear_modulus_voigt (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_shear_modulus_vrh (*burnman.AnisotropicMaterial property*), 298
isentropic_shear_modulus_vrh (*burnman.AnisotropicMineral property*), 322
isentropic_shear_modulus_vrh (*burnman.AnisotropicSolution property*), 521
isentropic_shear_modulus_vrh (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_stiffness_tensor (*burnman.AnisotropicMaterial property*), 297
isentropic_stiffness_tensor (*burnman.AnisotropicMineral property*), 316
isentropic_stiffness_tensor (*burnman.AnisotropicSolution property*), 521
isentropic_stiffness_tensor (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_thermal_gradient (*burnman.AnisotropicMaterial property*), 304
isentropic_thermal_gradient (*burnman.AnisotropicMineral property*), 323
isentropic_thermal_gradient (*burnman.AnisotropicSolution property*), 521
isentropic_thermal_gradient (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
isentropic_thermal_gradient (*burnman.CombinedMineral property*), 265
isentropic_thermal_gradient (*burnman.Composite property*), 339
isentropic_thermal_gradient (*burnman.ElasticSolution property*), 285
isentropic_thermal_gradient (*burnman.Material property*), 233
isentropic_thermal_gradient (*burnman.Mineral property*), 260
isentropic_thermal_gradient (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 776
isentropic_thermal_gradient (*burnman.PerplexMaterial property*), 245
isentropic_thermal_gradient (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_thermal_gradient (*burnman.RelaxedComposite property*), 347
isentropic_thermal_gradient (*burnman.RelaxedSolution property*), 541
isentropic_thermal_gradient (*burnman.Solution property*), 276
isentropic_universal_elastic_anisotropy (*burnman.AnisotropicMaterial property*), 299
isentropic_universal_elastic_anisotropy (*burnman.AnisotropicMineral property*), 323
isentropic_universal_elastic_anisotropy (*burnman.AnisotropicSolution property*), 521
isentropic_universal_elastic_anisotropy (*burnman.RelaxedAnisotropicSolution property*), 558
isentropic_youngs_modulus() (*burnman.AnisotropicMaterial method*), 299
isentropic_youngs_modulus() (*burnman.AnisotropicMineral method*), 323
isentropic_youngs_modulus() (*burnman.AnisotropicSolution method*), 522
isentropic_youngs_modulus() (*burnman.RelaxedAnisotropicSolution method*), 558
isothermal_bulk_modulus() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
isothermal_bulk_modulus_reuss (*burnman.AnisotropicMaterial property*), 304
isothermal_bulk_modulus_reuss (*burn-*

man.AnisotropicMineral property), 314
isothermal_bulk_modulus_reuss (*burnman.AnisotropicSolution property*), 522
isothermal_bulk_modulus_reuss (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
isothermal_bulk_modulus_reuss (*burnman.CombinedMineral property*), 261
isothermal_bulk_modulus_reuss (*burnman.Composite property*), 333
isothermal_bulk_modulus_reuss (*burnman.ElasticSolution property*), 281
isothermal_bulk_modulus_reuss (*burnman.Layer property*), 634
isothermal_bulk_modulus_reuss (*burnman.Material property*), 229
isothermal_bulk_modulus_reuss (*burnman.Mineral property*), 250
isothermal_bulk_modulus_reuss (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
isothermal_bulk_modulus_reuss (*burnman.PerplexMaterial property*), 236
isothermal_bulk_modulus_reuss (*burnman.Planet property*), 642
isothermal_bulk_modulus_reuss (*burnman.RelaxedAnisotropicSolution property*), 548
isothermal_bulk_modulus_reuss (*burnman.RelaxedComposite property*), 343
isothermal_bulk_modulus_reuss (*burnman.RelaxedSolution property*), 536
isothermal_bulk_modulus_reuss (*burnman.Solution property*), 271
isothermal_bulk_modulus_reuss() (*burnman.eos.aa.AA method*), 408
isothermal_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
isothermal_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BM3 method*), 362
isothermal_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 366
isothermal_bulk_modulus_reuss() (*burnman.eos.birch_murnaghan.BM4 method*), 369
isothermal_bulk_modulus_reuss() (*burnman.eos.brosh_calphad.BroshCalphad method*), 412
isothermal_bulk_modulus_reuss() (*burnman.eos.cork.CORK method*), 416
isothermal_bulk_modulus_reuss() (*burnman.eos.dks_liquid.DKS_L method*), 421
isothermal_bulk_modulus_reuss() (*burnman.eos.dks_solid.DKS_S method*), 424
isothermal_bulk_modulus_reuss() (*burnman.eos.EquationOfState method*), 402
isothermal_bulk_modulus_reuss() (*burnman.eos.hp.HP98 method*), 428
isothermal_bulk_modulus_reuss() (*burnman.eos.hp.HP_TMT method*), 432
isothermal_bulk_modulus_reuss() (*burnman.eos.hp.HP_TMTL method*), 436
isothermal_bulk_modulus_reuss() (*burnman.eos.IsothermalEquationOfState method*), 357
isothermal_bulk_modulus_reuss() (*burnman.eos.macaw.MACAW method*), 375
isothermal_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 441
isothermal_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 446
isothermal_bulk_modulus_reuss() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
isothermal_bulk_modulus_reuss() (*burnman.eos.modified_tait.MT method*), 379
isothermal_bulk_modulus_reuss() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 451
isothermal_bulk_modulus_reuss() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 456
isothermal_bulk_modulus_reuss() (*burnman.eos.morse_potential.Morse method*), 383
isothermal_bulk_modulus_reuss() (*burnman.eos.murnaghan.Murnaghan method*), 387
isothermal_bulk_modulus_reuss() (*burnman.eos.reciprocal_kprime.RKprime*

method), 390
isothermal_bulk_modulus_reuss() (*burnman.eos.slb.SLB2 method*), 462
isothermal_bulk_modulus_reuss() (*burnman.eos.slb.SLB3 method*), 467
isothermal_bulk_modulus_reuss() (*burnman.eos.slb.SLB3Conductive method*), 471
isothermal_bulk_modulus_reuss() (*burnman.eos.slb.SLBBase method*), 472
isothermal_bulk_modulus_reuss() (*burnman.eos.spock.SPOCK method*), 394
isothermal_bulk_modulus_reuss() (*burnman.eos.vinet.Vinet method*), 398
isothermal_bulk_modulus_voigt (*burnman.AnisotropicMineral property*), 315
isothermal_bulk_modulus_voigt (*burnman.AnisotropicSolution property*), 522
isothermal_bulk_modulus_voigt (*burnman.RelaxedAnisotropicSolution property*), 559
isothermal_compliance_tensor (*burnman.AnisotropicMineral property*), 315
isothermal_compliance_tensor (*burnman.AnisotropicSolution property*), 522
isothermal_compliance_tensor (*burnman.RelaxedAnisotropicSolution property*), 547
isothermal_compressibility_reuss (*burnman.AnisotropicMaterial property*), 304
isothermal_compressibility_reuss (*burnman.AnisotropicMineral property*), 315
isothermal_compressibility_reuss (*burnman.AnisotropicSolution property*), 522
isothermal_compressibility_reuss (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
isothermal_compressibility_reuss (*burnman.CombinedMineral property*), 262
isothermal_compressibility_reuss (*burnman.Composite property*), 334
isothermal_compressibility_reuss (*burnman.ElasticSolution property*), 281
isothermal_compressibility_reuss (*burnman.Layer property*), 634
isothermal_compressibility_reuss (*burnman.Material property*), 229
isothermal_compressibility_reuss (*burnman.Mineral property*), 254
isothermal_compressibility_reuss (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 776
isothermal_compressibility_reuss (*burnman.PerplexMaterial property*), 241
isothermal_compressibility_reuss (*burnman.Planet property*), 643
isothermal_compressibility_reuss (*burnman.RelaxedAnisotropicSolution property*), 547
isothermal_compressibility_reuss (*burnman.RelaxedComposite property*), 343
isothermal_compressibility_reuss (*burnman.RelaxedSolution property*), 535
isothermal_compressibility_reuss (*burnman.Solution property*), 271
isothermal_compressibility_tensor (*burnman.AnisotropicMineral property*), 317
isothermal_compressibility_tensor (*burnman.AnisotropicSolution property*), 522
isothermal_compressibility_tensor (*burnman.RelaxedAnisotropicSolution property*), 559
isothermal_compressibility_voigt (*burnman.AnisotropicMineral property*), 315
isothermal_compressibility_voigt (*burnman.AnisotropicSolution property*), 522
isothermal_compressibility_voigt (*burnman.RelaxedAnisotropicSolution property*), 559
isothermal_stiffness_tensor (*burnman.AnisotropicMineral property*), 316
isothermal_stiffness_tensor (*burnman.AnisotropicSolution property*), 522
isothermal_stiffness_tensor (*burnman.RelaxedAnisotropicSolution property*), 546
IsothermalEquationOfState (*class in burnman.eos*), 355
items() (*burnman.utils.misc.OrderedCounter method*), 841
Iterates (*class in burnman.optimize.nonlinear_solvers*), 798
IUPAC_element_order (*in module burnman.utils.chemistry*), 827

J

J (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
jacobian() (*in module burnman.tools.equilibration*), 804

K

K() (*burnman.classes.seismic.AK135 method*), 682
K() (*burnman.classes.seismic.Fast method*), 672
K() (*burnman.classes.seismic.IASP91 method*), 678
K() (*burnman.classes.seismic.PREM method*), 666
K() (*burnman.classes.seismic.Seismic1DModel method*), 661
K() (*burnman.classes.seismic.SeismicTable method*), 665
K() (*burnman.classes.seismic.Slow method*), 669
K() (*burnman.classes.seismic.STW105 method*), 675
K_eff (*burnman.AnisotropicMaterial property*), 301
K_eff (*burnman.AnisotropicMineral property*), 318
K_eff (*burnman.AnisotropicSolution property*), 513
K_eff (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
K_eff (*burnman.CombinedMineral property*), 262
K_eff (*burnman.Composite property*), 337
K_eff (*burnman.ElasticSolution property*), 283
K_eff (*burnman.Material property*), 234
K_eff (*burnman.Mineral property*), 256
K_eff (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
K_eff (*burnman.PerplexMaterial property*), 242
K_eff (*burnman.RelaxedAnisotropicSolution property*), 550
K_eff (*burnman.RelaxedComposite property*), 344
K_eff (*burnman.RelaxedSolution property*), 537
K_eff (*burnman.Solution property*), 274
K_S (*burnman.AnisotropicMaterial property*), 300
K_S (*burnman.AnisotropicMineral property*), 314
K_S (*burnman.AnisotropicSolution property*), 512
K_S (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
K_S (*burnman.CombinedMineral property*), 262
K_S (*burnman.Composite property*), 337
K_S (*burnman.ElasticSolution property*), 283
K_S (*burnman.Layer property*), 637
K_S (*burnman.Material property*), 234
K_S (*burnman.Mineral property*), 256
K_S (*burnman.optimize.composition_fitting.DummyCompositionSolu property*), 771
K_S (*burnman.PerplexMaterial property*), 242
K_S (*burnman.Planet property*), 645
K_S (*burnman.RelaxedAnisotropicSolution property*), 549
K_S (*burnman.RelaxedComposite property*), 343
K_S (*burnman.RelaxedSolution property*), 536
K_S (*burnman.Solution property*), 274
K_T (*burnman.AnisotropicMaterial property*), 300
K_T (*burnman.AnisotropicMineral property*), 313
K_T (*burnman.AnisotropicSolution property*), 513
K_T (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
K_T (*burnman.CombinedMineral property*), 262
K_T (*burnman.Composite property*), 337
K_T (*burnman.ElasticSolution property*), 283
K_T (*burnman.Layer property*), 637
K_T (*burnman.Material property*), 234
K_T (*burnman.Mineral property*), 256
K_T (*burnman.optimize.composition_fitting.DummyCompositionSolu property*), 771
K_T (*burnman.PerplexMaterial property*), 242
K_T (*burnman.Planet property*), 645
K_T (*burnman.RelaxedAnisotropicSolution property*), 550
K_T (*burnman.RelaxedComposite property*), 343
K_T (*burnman.RelaxedSolution property*), 536
K_T (*burnman.Solution property*), 274
Katsura2022 (*in burnman.classes.geotherm*), 625
keys() (*burnman.utils.misc.OrderedCounter method*), 841
kinematics() (*in module examples.example_hyperelastic_stress_rates*), 188
KToverV() (*in module burnman.eos.bukowinski_electronic*), 480

L

l2_norm_profile() (*in module burnman.utils.math*), 838
l2_norm_profiles() (*in module burnman.utils.math*), 837
lambda_bounds() (*in module burnman.tools.equilibration*), 805
landau_excesses() (*in module burnman.eos.property_modifiers*), 486

landau_hp_excesses() (*in module burnman.eos.property_modifiers*), 487
landau_slb_2022_excesses() (*in module burnman.eos.property_modifiers*), 488
Layer (*class in burnman*), 629
limits (*burnman.MaterialPolytope property*), 612
linear_excesses() (*in module burnman.eos.property_modifiers*), 488
linear_interpol() (*in module burnman.utils.math*), 835
lmda (*burnman.optimize.nonlinear_solvers.Iterates attribute*), 798
lnVoverV0_approx() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 456
LogNormal() (*in module burnman.eos.anharmonic_thermal_models*), 478
lookup_and_interpolate() (*in module burnman.utils.misc*), 843

M

MACAW (*class in burnman.eos.macaw*), 375
magnetic_excesses_chs() (*in module burnman.eos.property_modifiers*), 489
make() (*burnman.Layer method*), 631
make() (*burnman.Planet method*), 640
make_params() (*in module burnman.eos.macaw*), 482
mass (*burnman.AnisotropicMaterial property*), 305
mass (*burnman.AnisotropicMineral property*), 323
mass (*burnman.AnisotropicSolution property*), 523
mass (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
mass (*burnman.CombinedMineral property*), 265
mass (*burnman.Composite property*), 339
mass (*burnman.ElasticSolution property*), 285
mass (*burnman.Layer property*), 631
mass (*burnman.Material property*), 228
mass (*burnman.Mineral property*), 258
mass (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
mass (*burnman.PerplexMaterial property*), 245
mass (*burnman.Planet property*), 640
mass (*burnman.RelaxedAnisotropicSolution property*), 559
mass (*burnman.RelaxedComposite property*), 347
mass (*burnman.RelaxedSolution property*), 541
mass (*burnman.Solution property*), 276
Material (*class in burnman*), 223
MaterialPolytope (*class in burnman*), 612
MAX_ITERATIONS (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
MechanicalSolution (*class in burnman.classes.solutionmodel*), 564
merge_two_dicts() (*in module burnman.utils.misc*), 842
MGD2 (*class in burnman.eos.mie_grueneisen_debye*), 439
MGD3 (*class in burnman.eos.mie_grueneisen_debye*), 442
MGDBase (*class in burnman.eos.mie_grueneisen_debye*), 447
mineos_formatted_data_and_reference() (*in module burnman.tools.output_seismo*), 823
Mineral (*class in burnman*), 248
MineralFit (*class in burnman.optimize.eos_fitting*), 783
MineralFitV (*class in burnman.optimize.eos_fitting*), 786
mle_estimate() (*in module burnman.optimize.nonlinear_fitting*), 792
ModularMGD (*class in burnman.eos.modular_mie_grueneisen_debye*), 450
ModularMGDWithAnharmonicity (*class in burnman.eos.modular_mie_grueneisen_debye*), 455
module
    burnman, 1
    burnman.calibrants, 762
    burnman.classes.geotherm, 623
    burnman.minerals, 685
    burnman.optimize.composition_fitting, 771
    burnman.optimize.eos_fitting, 783
    burnman.optimize.linear_fitting, 790
    burnman.optimize.nonlinear_fitting, 791
    burnman.optimize.nonlinear_solvers, 798
    burnman.tools.chemistry, 816
    burnman.tools.eos, 813
    burnman.tools.equilibration, 802
    burnman.tools.output_seismo, 821

burnman.tools.partitioning, 819
burnman.tools.plot, 820
burnman.tools.polytope, 823
burnman.tools.seismic, 821
burnman.tools.solution, 815
burnman.tools.thermobarometry, 809
burnman.tools.unitcell, 802
burnman.utils.anisotropy, 833
burnman.utils.chemistry, 828
burnman.utils.math, 835
burnman.utils.misc, 840
burnman.utils.plotting, 834
burnman.utils.reductions, 840
burnman.utils.unitcell, 826
contrib.CHRU2014.paper_averaging, 220
contrib.CHRU2014.paper_benchmark, 220
contrib.CHRU2014.paper_fit_data, 220
contrib.CHRU2014.paper_incorrect_averaging, 220
contrib.CHRU2014.paper_onefit, 220
contrib.CHRU2014.paper_opt_pv, 220
contrib.CHRU2014.paper_uncertain, 221
contrib.cider_tutorial_2014.step_1, 57
contrib.cider_tutorial_2014.step_2, 58
contrib.cider_tutorial_2014.step_3, 59
examples.example_anisotropic_mineral, 175
examples.example_anisotropy, 174
examples.example_averaging, 183
examples.example_beginner, 179
examples.example_build_planet, 194
examples.example_calibrants, 174
examples.example_chemical_potentials, 184
examples.example_compare_all_methods, 193
examples.example_composite, 172
examples.example_composite_seismic_velocities, 182
examples.example_composition, 178
examples.example_equilibrate, 207
examples.example_fit_composition, 196
examples.example_fit_data, 197
examples.example_fit_eos, 200
examples.example_fit_solution, 207
examples.example_geotherms, 176
examples.example_gibbs_modifiers, 166
examples.example_grid, 221
examples.example_hyperelastic_stress_rates, 186
examples.example_mineral, 165
examples.example_olivine_binary, 219
examples.example_optimize_pv, 192
examples.example_seismic, 180
examples.example_solution, 169
examples.example_spintransition, 190
examples.example_spintransition_thermal, 191
examples.example_user_input_material, 192
examples.example_woutput, 221
molar_composition (*burnman.Composition property*), 609
molar_enthalpy (*burnman.AnisotropicMaterial property*), 305
molar_enthalpy (*burnman.AnisotropicMineral property*), 323
molar_enthalpy (*burnman.AnisotropicSolution property*), 523
molar_enthalpy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
molar_enthalpy (*burnman.CombinedMineral property*), 261
molar_enthalpy (*burnman.Composite property*), 333
molar_enthalpy (*burnman.ElasticSolution property*), 281
molar_enthalpy (*burnman.Layer property*), 634
molar_enthalpy (*burnman.Material property*), 229
molar_enthalpy (*burnman.Mineral property*), 253
molar_enthalpy (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 776
molar_enthalpy (*burnman.PerplexMaterial property*), 236
molar_enthalpy (*burnman.Planet property*), 642
molar_enthalpy (*burnman.RelaxedAnisotropicSolution property*), 559
molar_enthalpy (*burnman.RelaxedComposite property*), 348

molar_enthalpy (*burnman.RelaxedSolution property*), 541
molar_enthalpy (*burnman.Solution property*), 271
molar_entropy (*burnman.AnisotropicMaterial property*), 305
molar_entropy (*burnman.AnisotropicMineral property*), 324
molar_entropy (*burnman.AnisotropicSolution property*), 523
molar_entropy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
molar_entropy (*burnman.CombinedMineral property*), 261
molar_entropy (*burnman.Composite property*), 333
molar_entropy (*burnman.ElasticSolution property*), 281
molar_entropy (*burnman.Layer property*), 633
molar_entropy (*burnman.Material property*), 228
molar_entropy (*burnman.Mineral property*), 250
molar_entropy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 776
molar_entropy (*burnman.PerplexMaterial property*), 236
molar_entropy (*burnman.Planet property*), 642
molar_entropy (*burnman.RelaxedAnisotropicSolution property*), 559
molar_entropy (*burnman.RelaxedComposite property*), 348
molar_entropy (*burnman.RelaxedSolution property*), 541
molar_entropy (*burnman.Solution property*), 271
molar_gibbs (*burnman.AnisotropicMaterial property*), 305
molar_gibbs (*burnman.AnisotropicMineral property*), 324
molar_gibbs (*burnman.AnisotropicSolution property*), 523
molar_gibbs (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
molar_gibbs (*burnman.CombinedMineral property*), 261
molar_gibbs (*burnman.Composite property*), 333
molar_gibbs (*burnman.ElasticSolution property*), 280
molar_gibbs (*burnman.Layer property*), 632
molar_gibbs (*burnman.Material property*), 227
molar_gibbs (*burnman.Mineral property*), 249
molar_gibbs (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_gibbs (*burnman.PerplexMaterial property*), 239
molar_gibbs (*burnman.Planet property*), 641
molar_gibbs (*burnman.RelaxedAnisotropicSolution property*), 559
molar_gibbs (*burnman.RelaxedComposite property*), 348
molar_gibbs (*burnman.RelaxedSolution property*), 541
molar_gibbs (*burnman.Solution property*), 270
molar_heat_capacity_p (*burnman.AnisotropicMaterial property*), 306
molar_heat_capacity_p (*burnman.AnisotropicMineral property*), 324
molar_heat_capacity_p (*burnman.AnisotropicSolution property*), 523
molar_heat_capacity_p (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292
molar_heat_capacity_p (*burnman.CombinedMineral property*), 261
molar_heat_capacity_p (*burnman.Composite property*), 334
molar_heat_capacity_p (*burnman.ElasticSolution property*), 281
molar_heat_capacity_p (*burnman.Layer property*), 636
molar_heat_capacity_p (*burnman.Material property*), 233
molar_heat_capacity_p (*burnman.Mineral property*), 251
molar_heat_capacity_p (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_heat_capacity_p (*burnman.PerplexMaterial property*), 237
molar_heat_capacity_p (*burnman.Planet property*), 645
molar_heat_capacity_p (*burn-*

man.RelaxedAnisotropicSolution property), 549
molar_heat_capacity_p (*burnman.RelaxedComposite property*), 343
molar_heat_capacity_p (*burnman.RelaxedSolution property*), 536
molar_heat_capacity_p (*burnman.Solution property*), 272
molar_heat_capacity_p() (*burnman.eos.aa.AA method*), 409
molar_heat_capacity_p() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 374
molar_heat_capacity_p() (*burnman.eos.birch_murnaghan.BM3 method*), 362
molar_heat_capacity_p() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 366
molar_heat_capacity_p() (*burnman.eos.birch_murnaghan.BM4 method*), 371
molar_heat_capacity_p() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
molar_heat_capacity_p() (*burnman.eos.cork.CORK method*), 416
molar_heat_capacity_p() (*burnman.eos.dks_liquid.DKS_L method*), 421
molar_heat_capacity_p() (*burnman.eos.dks_solid.DKS_S method*), 424
molar_heat_capacity_p() (*burnman.eos.EquationOfState method*), 404
molar_heat_capacity_p() (*burnman.eos.hp.HP98 method*), 428
molar_heat_capacity_p() (*burnman.eos.hp.HP_TMT method*), 433
molar_heat_capacity_p() (*burnman.eos.hp.HP_TMTL method*), 436
molar_heat_capacity_p() (*burnman.eos.IsothermalEquationOfState method*), 355
molar_heat_capacity_p() (*burnman.eos.macaw.MACAW method*), 378
molar_heat_capacity_p() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 441
molar_heat_capacity_p() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 446
molar_heat_capacity_p() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
molar_heat_capacity_p() (*burnman.eos.modified_tait.MT method*), 381
molar_heat_capacity_p() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
molar_heat_capacity_p() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 459
molar_heat_capacity_p() (*burnman.eos.morse_potential.Morse method*), 385
molar_heat_capacity_p() (*burnman.eos.murnaghan.Murnaghan method*), 389
molar_heat_capacity_p() (*burnman.eos.reciprocal_kprime.RKprime method*), 392
molar_heat_capacity_p() (*burnman.eos.slb.SLB2 method*), 462
molar_heat_capacity_p() (*burnman.eos.slb.SLB3 method*), 467
molar_heat_capacity_p() (*burnman.eos.slb.SLB3Conductive method*), 471
molar_heat_capacity_p() (*burnman.eos.slb.SLBBase method*), 472
molar_heat_capacity_p() (*burnman.eos.spock.SPOCK method*), 396
molar_heat_capacity_p() (*burnman.eos.vinet.Vinet method*), 400
molar_heat_capacity_v (*burnman.AnisotropicMaterial property*), 306
molar_heat_capacity_v (*burnman.AnisotropicMineral property*), 317
molar_heat_capacity_v (*burnman.AnisotropicSolution property*), 523
molar_heat_capacity_v (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 292

molar_heat_capacity_v (*burnman.CombinedMineral property*), 262
molar_heat_capacity_v (*burnman.Composite property*), 334
molar_heat_capacity_v (*burnman.ElasticSolution property*), 281
molar_heat_capacity_v (*burnman.Layer property*), 636
molar_heat_capacity_v (*burnman.Material property*), 232
molar_heat_capacity_v (*burnman.Mineral property*), 259
molar_heat_capacity_v (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_heat_capacity_v (*burnman.PerplexMaterial property*), 241
molar_heat_capacity_v (*burnman.Planet property*), 644
molar_heat_capacity_v (*burnman.RelaxedAnisotropicSolution property*), 559
molar_heat_capacity_v (*burnman.RelaxedComposite property*), 348
molar_heat_capacity_v (*burnman.RelaxedSolution property*), 536
molar_heat_capacity_v (*burnman.Solution property*), 272
molar_heat_capacity_v() (*burnman.eos.aa.AA method*), 411
molar_heat_capacity_v() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 374
molar_heat_capacity_v() (*burnman.eos.birch_murnaghan.BM3 method*), 362
molar_heat_capacity_v() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 367
molar_heat_capacity_v() (*burnman.eos.birch_murnaghan.BM4 method*), 371
molar_heat_capacity_v() (*burnman.eos.brosh_calphad.BroshCalphad method*), 415
molar_heat_capacity_v() (*burnman.eos.cork.CORK method*), 418
molar_heat_capacity_v() (*burnman.eos.dks_liquid.DKS_L method*), 423
molar_heat_capacity_v() (*burnman.eos.dks_solid.DKS_S method*), 427
molar_heat_capacity_v() (*burnman.eos.EquationOfState method*), 403
molar_heat_capacity_v() (*burnman.eos.hp.HP98 method*), 430
molar_heat_capacity_v() (*burnman.eos.hp.HP_TMT method*), 435
molar_heat_capacity_v() (*burnman.eos.hp.HP_TMTL method*), 438
molar_heat_capacity_v() (*burnman.eos.IsothermalEquationOfState method*), 357
molar_heat_capacity_v() (*burnman.eos.macaw.MACAW method*), 378
molar_heat_capacity_v() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 441
molar_heat_capacity_v() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 446
molar_heat_capacity_v() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 449
molar_heat_capacity_v() (*burnman.eos.modified_tait.MT method*), 382
molar_heat_capacity_v() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 454
molar_heat_capacity_v() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 459
molar_heat_capacity_v() (*burnman.eos.morse_potential.Morse method*), 385
molar_heat_capacity_v() (*burnman.eos.murnaghan.Murnaghan method*), 389
molar_heat_capacity_v() (*burnman.eos.reciprocal_kprime.RKprime method*), 393
molar_heat_capacity_v() (*burnman.eos.slb.SLB2 method*), 462
molar_heat_capacity_v() (*burn-*

man.eos.slb.SLB3 method), 467
molar_heat_capacity_v() (*burnman.eos.slb.SLB3Conductive method*), 471
molar_heat_capacity_v() (*burnman.eos.slb.SLBBase method*), 474
molar_heat_capacity_v() (*burnman.eos.spock.SPOCK method*), 396
molar_heat_capacity_v() (*burnman.eos.vinet.Vinet method*), 400
molar_heat_capacity_v() (*in module burnman.eos.debye*), 481
molar_heat_capacity_v() (*in module burnman.eos.einstein*), 482
molar_helmholtz (*burnman.AnisotropicMaterial property*), 306
molar_helmholtz (*burnman.AnisotropicMineral property*), 324
molar_helmholtz (*burnman.AnisotropicSolution property*), 523
molar_helmholtz (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
molar_helmholtz (*burnman.CombinedMineral property*), 261
molar_helmholtz (*burnman.Composite property*), 333
molar_helmholtz (*burnman.ElasticSolution property*), 280
molar_helmholtz (*burnman.Layer property*), 633
molar_helmholtz (*burnman.Material property*), 227
molar_helmholtz (*burnman.Mineral property*), 253
molar_helmholtz (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_helmholtz (*burnman.PerplexMaterial property*), 240
molar_helmholtz (*burnman.Planet property*), 641
molar_helmholtz (*burnman.RelaxedAnisotropicSolution property*), 559
molar_helmholtz (*burnman.RelaxedComposite property*), 348
molar_helmholtz (*burnman.RelaxedSolution property*), 541
molar_helmholtz (*burnman.Solution property*), 270
molar_internal_energy (*burnman.AnisotropicMaterial property*), 307
molar_internal_energy (*burnman.AnisotropicMineral property*), 325
molar_internal_energy (*burnman.AnisotropicSolution property*), 523
molar_internal_energy (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
molar_internal_energy (*burnman.CombinedMineral property*), 261
molar_internal_energy (*burnman.Composite property*), 333
molar_internal_energy (*burnman.ElasticSolution property*), 280
molar_internal_energy (*burnman.Layer property*), 632
molar_internal_energy (*burnman.Material property*), 226
molar_internal_energy (*burnman.Mineral property*), 252
molar_internal_energy (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_internal_energy (*burnman.PerplexMaterial property*), 240
molar_internal_energy (*burnman.Planet property*), 641
molar_internal_energy (*burnman.RelaxedAnisotropicSolution property*), 559
molar_internal_energy (*burnman.RelaxedComposite property*), 348
molar_internal_energy (*burnman.RelaxedSolution property*), 542
molar_internal_energy (*burnman.Solution property*), 270
molar_internal_energy() (*burnman.eos.aa.AA method*), 411
molar_internal_energy() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 375
molar_internal_energy() (*burnman.eos.birch_murnaghan.BM3 method*), 363
molar_internal_energy() (*burn-*

man.eos.birch_murnaghan.BM3Shear2 method), 367
molar_internal_energy() (*burnman.eos.birch_murnaghan.BM4 method*), 371
molar_internal_energy() (*burnman.eos.brosh_calphad.BroshCalphad method*), 415
molar_internal_energy() (*burnman.eos.cork.CORK method*), 419
molar_internal_energy() (*burnman.eos.dks_liquid.DKS_L method*), 423
molar_internal_energy() (*burnman.eos.dks_solid.DKS_S method*), 427
molar_internal_energy() (*burnman.eos.EquationOfState method*), 406
molar_internal_energy() (*burnman.eos.hp.HP98 method*), 430
molar_internal_energy() (*burnman.eos.hp.HP_TMT method*), 435
molar_internal_energy() (*burnman.eos.hp.HP_TMTL method*), 438
molar_internal_energy() (*burnman.eos.IsothermalEquationOfState method*), 358
molar_internal_energy() (*burnman.eos.macaw.MACAW method*), 378
molar_internal_energy() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
molar_internal_energy() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 446
molar_internal_energy() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 450
molar_internal_energy() (*burnman.eos.modified_tait.MT method*), 382
molar_internal_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 454
molar_internal_energy() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWithAnharmonicity method*), 459
molar_internal_energy() (*burnman.eos.morse_potential.Morse method*), 386
molar_internal_energy() (*burnman.eos.murnaghan.Murnaghan method*), 389
molar_internal_energy() (*burnman.eos.reciprocal_kprime.RKprime method*), 393
molar_internal_energy() (*burnman.eos.slb.SLB2 method*), 462
molar_internal_energy() (*burnman.eos.slb.SLB3 method*), 467
molar_internal_energy() (*burnman.eos.slb.SLB3Conductive method*), 471
molar_internal_energy() (*burnman.eos.slb.SLBBase method*), 475
molar_internal_energy() (*burnman.eos.spock.SPOCK method*), 397
molar_internal_energy() (*burnman.eos.vinet.Vinet method*), 400
molar_isometric_heat_capacity (*burnman.AnisotropicMineral property*), 329
molar_isometric_heat_capacity (*burnman.AnisotropicSolution property*), 523
molar_isometric_heat_capacity (*burnman.RelaxedAnisotropicSolution property*), 547
molar_mass (*burnman.AnisotropicMaterial property*), 307
molar_mass (*burnman.AnisotropicMineral property*), 325
molar_mass (*burnman.AnisotropicSolution property*), 523
molar_mass (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
molar_mass (*burnman.CombinedMineral property*), 261
molar_mass (*burnman.Composite property*), 333
molar_mass (*burnman.ElasticSolution property*), 280
molar_mass (*burnman.Layer property*), 633
molar_mass (*burnman.Material property*), 227
molar_mass (*burnman.Mineral property*), 252
molar_mass (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777

molar_mass (*burnman.PerplexMaterial property*), 239
molar_mass (*burnman.Planet property*), 641
molar_mass (*burnman.RelaxedAnisotropicSolution property*), 560
molar_mass (*burnman.RelaxedComposite property*), 348
molar_mass (*burnman.RelaxedSolution property*), 542
molar_mass (*burnman.Solution property*), 271
molar_volume (*burnman.AnisotropicMaterial property*), 307
molar_volume (*burnman.AnisotropicMineral property*), 325
molar_volume (*burnman.AnisotropicSolution property*), 523
molar_volume (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
molar_volume (*burnman.CombinedMineral property*), 261
molar_volume (*burnman.Composite property*), 333
molar_volume (*burnman.ElasticSolution property*), 280
molar_volume (*burnman.Layer property*), 633
molar_volume (*burnman.Material property*), 228
molar_volume (*burnman.Mineral property*), 250
molar_volume (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
molar_volume (*burnman.PerplexMaterial property*), 235
molar_volume (*burnman.Planet property*), 642
molar_volume (*burnman.RelaxedAnisotropicSolution property*), 560
molar_volume (*burnman.RelaxedComposite property*), 348
molar_volume (*burnman.RelaxedSolution property*), 542
molar_volume (*burnman.Solution property*), 271
molar_volume_from_unit_cell_volume() (*in module burnman.utils.unitcell*), 826
moment_of_inertia (*burnman.Layer property*), 631
moment_of_inertia (*burnman.Planet property*), 640
moment_of_inertia_factor (*burnman.Planet property*), 640
Morse (*class in burnman.eos.morse_potential*), 382
morse_potential() (*in module burnman.eos.morse_potential*), 483
most_common() (*burnman.utils.misc.OrderedCounter method*), 841
move_to_end() (*burnman.utils.misc.OrderedCounter method*), 841
MT (*class in burnman.eos.modified_tait*), 379
Murnaghan (*class in burnman.eos.murnaghan*), 386

N

n_elements (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
n_elements (*burnman.Composite property*), 336
n_elements (*burnman.RelaxedComposite property*), 348
n_endmembers (*burnman.AnisotropicSolution property*), 523
n_endmembers (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
n_endmembers (*burnman.Composite property*), 336
n_endmembers (*burnman.ElasticSolution property*), 282
n_endmembers (*burnman.MaterialPolytope property*), 612
n_endmembers (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
n_endmembers (*burnman.RelaxedAnisotropicSolution property*), 560
n_endmembers (*burnman.RelaxedComposite property*), 348
n_endmembers (*burnman.RelaxedSolution property*), 542
n_endmembers (*burnman.Solution property*), 273
n_it (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
n_reactions (*burnman.AnisotropicSolution property*), 523
n_reactions (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293

n_reactions (*burnman.Composite property*), 336
n_reactions (*burnman.ElasticSolution property*), 282
n_reactions (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
n_reactions (*burnman.RelaxedAnisotropicSolution property*), 560
n_reactions (*burnman.RelaxedComposite property*), 348
n_reactions (*burnman.RelaxedSolution property*), 542
n_reactions (*burnman.Solution property*), 272
name (*burnman.AnisotropicMaterial property*), 307
name (*burnman.AnisotropicMineral property*), 312
name (*burnman.AnisotropicSolution property*), 524
name (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
name (*burnman.CombinedMineral property*), 265
name (*burnman.Composite property*), 332
name (*burnman.ElasticSolution property*), 279
name (*burnman.Material property*), 223
name (*burnman.Mineral property*), 248
name (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
name (*burnman.PerplexMaterial property*), 235
name (*burnman.RelaxedAnisotropicSolution property*), 560
name (*burnman.RelaxedComposite property*), 348
name (*burnman.RelaxedSolution property*), 542
name (*burnman.Solution property*), 268
nondimensional_d2helmholtz_dTheta2() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nondimensional_dentropy_dTheta() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nondimensional_dhelmholtz_dTheta() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nondimensional_entropy() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nondimensional_heat_capacity() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nondimensional_helmholtz_energy() (*burnman.eos.anharmonic_thermal_models.AnharmonicThermalModelBase method*), 477
nonlinear_least_squares_fit() (*in module burnman.optimize.nonlinear_fitting*), 793
nonlinear_least_squares_fit_differential_evolution() (*in module burnman.optimize.nonlinear_fitting*), 795
NonLinearModel (*class in burnman.optimize.nonlinear_fitting*), 791
normal() (*burnman.optimize.eos_fitting.MineralFit method*), 784
normal() (*burnman.optimize.eos_fitting.MineralFitV method*), 786
normal() (*burnman.optimize.eos_fitting.SolutionFit method*), 789
normal() (*burnman.optimize.nonlinear_fitting.NonLinearModel method*), 791
number_of_moles (*burnman.AnisotropicMaterial property*), 308
number_of_moles (*burnman.AnisotropicMineral property*), 326
number_of_moles (*burnman.AnisotropicSolution property*), 524
number_of_moles (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
number_of_moles (*burnman.CombinedMineral property*), 265
number_of_moles (*burnman.Composite property*), 339
number_of_moles (*burnman.ElasticSolution property*), 285
number_of_moles (*burnman.Material property*), 227
number_of_moles (*burnman.Mineral property*), 259
number_of_moles (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
number_of_moles (*burnman.PerplexMaterial property*), 245
number_of_moles (*burnman.RelaxedAnisotropicSolution property*), 560
number_of_moles (*burnman.RelaxedComposite property*), 352
number_of_moles (*burnman.RelaxedSolution property*), 542

number_of_moles (*burnman.Solution property*), 277

O

ones (*burnman.AnisotropicSolution property*), 512
ones (*burnman.classes.solutionmodel.AsymmetricRegularSolution property*), 581
ones (*burnman.classes.solutionmodel.FunctionSolution property*), 603
ones (*burnman.classes.solutionmodel.IdealSolution property*), 572
ones (*burnman.classes.solutionmodel.SubregularSolution property*), 596
ones (*burnman.classes.solutionmodel.SymmetricRegularSolution property*), 589
ones (*burnman.RelaxedAnisotropicSolution property*), 560
ordered_compositional_array() (*in module burnman.utils.chemistry*), 831
OrderedCounter (*class in burnman.utils.misc*), 840

P

P (*burnman.AnisotropicMaterial property*), 301
P (*burnman.AnisotropicMineral property*), 318
P (*burnman.AnisotropicSolution property*), 513
P (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
P (*burnman.CombinedMineral property*), 262
P (*burnman.Composite property*), 337
P (*burnman.ElasticSolution property*), 283
P (*burnman.Layer property*), 636
P (*burnman.Material property*), 233
P (*burnman.Mineral property*), 256
P (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 771
P (*burnman.PerplexMaterial property*), 242
P (*burnman.Planet property*), 645
P (*burnman.RelaxedAnisotropicSolution property*), 550
P (*burnman.RelaxedComposite property*), 344
P (*burnman.RelaxedSolution property*), 537
P (*burnman.Solution property*), 274
p_wave_velocity (*burnman.AnisotropicMaterial property*), 308
p_wave_velocity (*burnman.AnisotropicMineral property*), 326
p_wave_velocity (*burnman.AnisotropicSolution property*), 524
p_wave_velocity (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
p_wave_velocity (*burnman.CombinedMineral property*), 262
p_wave_velocity (*burnman.Composite property*), 334
p_wave_velocity (*burnman.ElasticSolution property*), 281
p_wave_velocity (*burnman.Layer property*), 635
p_wave_velocity (*burnman.Material property*), 231
p_wave_velocity (*burnman.Mineral property*), 254
p_wave_velocity (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
p_wave_velocity (*burnman.PerplexMaterial property*), 238
p_wave_velocity (*burnman.Planet property*), 643
p_wave_velocity (*burnman.RelaxedAnisotropicSolution property*), 560
p_wave_velocity (*burnman.RelaxedComposite property*), 348
p_wave_velocity (*burnman.RelaxedSolution property*), 542
p_wave_velocity (*burnman.Solution property*), 271
Pade() (*in module burnman.eos.anharmonic_thermal_models*), 478
partial_entropies (*burnman.AnisotropicSolution property*), 524
partial_entropies (*burnman.ElasticSolution property*), 280
partial_entropies (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
partial_entropies (*burnman.RelaxedAnisotropicSolution property*), 560
partial_entropies (*burnman.RelaxedSolution property*), 542
partial_entropies (*burnman.Solution property*), 270
partial_gibbs (*burnman.AnisotropicSolution property*), 524

partial_gibbs (*burnman.ElasticSolution property*), 280
partial_gibbs (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
partial_gibbs (*burnman.RelaxedAnisotropicSolution property*), 560
partial_gibbs (*burnman.RelaxedSolution property*), 542
partial_gibbs (*burnman.Solution property*), 270
partial_volumes (*burnman.AnisotropicSolution property*), 524
partial_volumes (*burnman.ElasticSolution property*), 280
partial_volumes (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 777
partial_volumes (*burnman.RelaxedAnisotropicSolution property*), 560
partial_volumes (*burnman.RelaxedSolution property*), 542
partial_volumes (*burnman.Solution property*), 270
PerplexMaterial (*class in burnman*), 235
phase_composition_constraints() (*in module burnman.tools.equilibration*), 805
phase_fraction_constraints() (*in module burnman.tools.equilibration*), 805
phase_masses (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
phase_masses (*burnman.Composite property*), 333
phase_masses (*burnman.RelaxedComposite property*), 349
phase_volumes (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 293
phase_volumes (*burnman.Composite property*), 333
phase_volumes (*burnman.RelaxedComposite property*), 349
Planet (*class in burnman*), 639
Plesa2022Mars6cm3 (*class in burnman.classes.geotherm*), 626
plot_cov_ellipse() (*in module burnman.optimize.nonlinear_fitting*), 796
plot_projected_elastic_properties() (*in module burnman.tools.plot*), 820
plot_residuals() (*in module burnman.optimize.nonlinear_fitting*), 798
polar_decomposition() (*in module burnman.utils.anisotropy*), 834
pop() (*burnman.utils.misc.OrderedCounter method*), 841
popitem() (*burnman.utils.misc.OrderedCounter method*), 841
PowerLaw (*class in burnman.eos.anharmonic_prefactor_models*), 476
PowerLawGamma() (*in module burnman.eos.debye_temperature_models*), 481
PowerLawGammaSimple() (*in module burnman.eos.debye_temperature_models*), 481
PREM (*class in burnman.classes.seismic*), 665
pressure (*burnman.AnisotropicMaterial property*), 308
pressure (*burnman.AnisotropicMineral property*), 326
pressure (*burnman.AnisotropicSolution property*), 524
pressure (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
pressure (*burnman.CombinedMineral property*), 265
pressure (*burnman.Composite property*), 339
pressure (*burnman.ElasticSolution property*), 286
pressure (*burnman.Layer property*), 632
pressure (*burnman.Material property*), 226
pressure (*burnman.Mineral property*), 259
pressure (*burnman.optimize.composition_fitting.DummyCompositio property*), 778
pressure (*burnman.PerplexMaterial property*), 245
pressure (*burnman.Planet property*), 641
pressure (*burnman.RelaxedAnisotropicSolution property*), 560
pressure (*burnman.RelaxedComposite property*), 349
pressure (*burnman.RelaxedSolution property*), 542
pressure (*burnman.Solution property*), 277
pressure() (*burnman.Calibrant method*), 352
pressure() (*burnman.classes.seismic.AK135 method*), 684

pressure() (*burnman.classes.seismic.Fast method*), 674
pressure() (*burnman.classes.seismic.IASP91 method*), 680
pressure() (*burnman.classes.seismic.PREM method*), 668
pressure() (*burnman.classes.seismic.Seismic1DModel method*), 660
pressure() (*burnman.classes.seismic.SeismicTable method*), 663
pressure() (*burnman.classes.seismic.Slow method*), 671
pressure() (*burnman.classes.seismic.STW105 method*), 677
pressure() (*burnman.eos.aa.AA method*), 408
pressure() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
pressure() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
pressure() (*burnman.eos.birch_murnaghan.BM3 method*), 363
pressure() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 367
pressure() (*burnman.eos.birch_murnaghan.BM4 method*), 368
pressure() (*burnman.eos.brosh_calphad.BroshCalphad method*), 412
pressure() (*burnman.eos.cork.CORK method*), 419
pressure() (*burnman.eos.dks_liquid.DKS_L method*), 420
pressure() (*burnman.eos.dks_solid.DKS_S method*), 424
pressure() (*burnman.eos.EquationOfState method*), 401
pressure() (*burnman.eos.hp.HP98 method*), 431
pressure() (*burnman.eos.hp.HP_TMT method*), 432
pressure() (*burnman.eos.hp.HP_TMTL method*), 439
pressure() (*burnman.eos.IsothermalEquationOfState method*), 358
pressure() (*burnman.eos.macaw.MACAW method*), 376
pressure() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
pressure() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 447
pressure() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
pressure() (*burnman.eos.modified_tait.MT method*), 379
pressure() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 451
pressure() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 456
pressure() (*burnman.eos.morse_potential.Morse method*), 383
pressure() (*burnman.eos.murnaghan.Murnaghan method*), 386
pressure() (*burnman.eos.reciprocal_kprime.RKprime method*), 390
pressure() (*burnman.eos.slb.SLB2 method*), 463
pressure() (*burnman.eos.slb.SLB3 method*), 468
pressure() (*burnman.eos.slb.SLB3Conductive method*), 472
pressure() (*burnman.eos.slb.SLBBase method*), 472
pressure() (*burnman.eos.spock.SPOCK method*), 394
pressure() (*burnman.eos.vinet.Vinet method*), 397
pressure() (*in module burnman.eos.bukowinski_electronic*), 480
pressure() (*in module burnman.eos.murnaghan*), 483
pressure_fourth_order() (*in module burnman.eos.birch_murnaghan*), 479
pressure_hessian() (*burnman.classes.elasticsolutionmodel.ElasticAsymmetricRegula method*), 583
pressure_hessian() (*burnman.classes.elasticsolutionmodel.ElasticFunctionSolution method*), 604

pressure_hessian() (*burnman.classes.elasticsolutionmodel.ElasticIdealSolution method*), 575
pressure_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSubregularSolution method*), 598
pressure_hessian() (*burnman.classes.elasticsolutionmodel.ElasticSymmetricRegularSolution method*), 591
pressure_modified_tait() (*in module burnman.eos.modified_tait*), 482
pressure_third_order() (*in module burnman.eos.birch_murnaghan*), 479
pressure_vinet() (*in module burnman.eos.vinet*), 490
pretty_plot() (*in module burnman.tools.plot*), 820
pretty_print_table() (*in module burnman.utils.misc*), 843
pretty_print_values() (*in module burnman.utils.misc*), 843
pretty_string_values() (*in module burnman.utils.misc*), 842
print() (*burnman.Composition method*), 609
print_minerals_of_current_state() (*burnman.AnisotropicMaterial method*), 308
print_minerals_of_current_state() (*burnman.AnisotropicMineral method*), 327
print_minerals_of_current_state() (*burnman.AnisotropicSolution method*), 524
print_minerals_of_current_state() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 294
print_minerals_of_current_state() (*burnman.CombinedMineral method*), 266
print_minerals_of_current_state() (*burnman.Composite method*), 340
print_minerals_of_current_state() (*burnman.ElasticSolution method*), 286
print_minerals_of_current_state() (*burnman.Material method*), 224
print_minerals_of_current_state() (*burnman.Mineral method*), 259
print_minerals_of_current_state() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 778
print_minerals_of_current_state() (*burnman.PerplexMaterial method*), 246
print_minerals_of_current_state() (*burnman.RelaxedAnisotropicSolution method*), 561
print_minerals_of_current_state() (*burnman.RelaxedComposite method*), 349
print_minerals_of_current_state() (*burnman.RelaxedSolution method*), 543
print_minerals_of_current_state() (*burnman.Solution method*), 277
process_eq_constraints() (*in module burnman.tools.equilibration*), 807
process_solution_chemistry() (*in module burnman.utils.chemistry*), 829

Q

QG() (*burnman.classes.seismic.AK135 method*), 682
QG() (*burnman.classes.seismic.Fast method*), 672
QG() (*burnman.classes.seismic.IASP91 method*), 679
QG() (*burnman.classes.seismic.PREM method*), 666
QG() (*burnman.classes.seismic.Seismic1DModel method*), 661
QG() (*burnman.classes.seismic.SeismicTable method*), 664
QG() (*burnman.classes.seismic.Slow method*), 669
QG() (*burnman.classes.seismic.STW105 method*), 676
QK() (*burnman.classes.seismic.AK135 method*), 682
QK() (*burnman.classes.seismic.Fast method*), 673
QK() (*burnman.classes.seismic.IASP91 method*), 679
QK() (*burnman.classes.seismic.PREM method*), 666
QK() (*burnman.classes.seismic.Seismic1DModel method*), 661
QK() (*burnman.classes.seismic.SeismicTable method*), 663
QK() (*burnman.classes.seismic.Slow method*), 669
QK() (*burnman.classes.seismic.STW105 method*), 676

R

radius() (*burnman.classes.seismic.AK135 method*), 684
radius() (*burnman.classes.seismic.Fast method*), 674
radius() (*burnman.classes.seismic.IASP91 method*), 681

radius() (*burnman.classes.seismic.PREM method*), 668
radius() (*burnman.classes.seismic.SeismicTable method*), 664
radius() (*burnman.classes.seismic.Slow method*), 671
radius() (*burnman.classes.seismic.STW105 method*), 677
raw_vertices (*burnman.MaterialPolytope property*), 612
reaction_affinities (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
reaction_affinities (*burnman.Composite property*), 335
reaction_affinities (*burnman.RelaxedComposite property*), 349
reaction_basis (*burnman.AnisotropicSolution property*), 524
reaction_basis (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
reaction_basis (*burnman.Composite property*), 336
reaction_basis (*burnman.ElasticSolution property*), 282
reaction_basis (*burnman.optimize.composition_fitting.DummyComposition property*), 778
reaction_basis (*burnman.RelaxedAnisotropicSolution property*), 561
reaction_basis (*burnman.RelaxedComposite property*), 349
reaction_basis (*burnman.RelaxedSolution property*), 543
reaction_basis (*burnman.Solution property*), 272
reaction_basis_as_strings (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
reaction_basis_as_strings (*burnman.Composite property*), 336
reaction_basis_as_strings (*burnman.RelaxedComposite property*), 349
reaction_matrix_as_strings() (*in module burnman.utils.chemistry*), 832
reactions_from_formulae() (*in module burnman.tools.chemistry*), 818
reactions_from_stoichiometric_matrix() (*in module burnman.tools.chemistry*), 818
read_table() (*in module burnman.utils.misc*), 843
reduced_stoichiometric_array (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
reduced_stoichiometric_array (*burnman.Composite property*), 336
reduced_stoichiometric_array (*burnman.RelaxedComposite property*), 349
relative_error() (*in module examples.example_hyperelastic_stress_rates*), 189
relative_fugacity() (*in module burnman.tools.chemistry*), 816
RelaxedAnisotropicSolution (*class in burnman*), 545
RelaxedComposite (*class in burnman*), 341
RelaxedSolution (*class in burnman*), 534
remove_null_components() (*burnman.Composition method*), 608
renormalize() (*burnman.Composition method*), 608
reorder_to_echelon() (*in module burnman.tools.polytope*), 824
reset() (*burnman.AnisotropicMaterial method*), 308
reset() (*burnman.AnisotropicMineral method*), 327
reset() (*burnman.AnisotropicSolution method*), 525
reset() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 294
reset() (*burnman.CombinedMineral method*), 266
reset() (*burnman.Composite method*), 340
reset() (*burnman.ElasticSolution method*), 286
reset() (*burnman.Layer method*), 630
reset() (*burnman.Material method*), 224
reset() (*burnman.Mineral method*), 259
reset() (*burnman.optimize.composition_fitting.DummyComposition method*), 778
reset() (*burnman.PerplexMaterial method*), 246
reset() (*burnman.Planet method*), 639
reset() (*burnman.RelaxedAnisotropicSolution method*), 561
reset() (*burnman.RelaxedComposite method*), 349
reset() (*burnman.RelaxedSolution method*), 543
reset() (*burnman.Solution method*), 277

Reuss (*class in burnman.averaging_schemes*), 618
rho (*burnman.AnisotropicMaterial property*), 308
rho (*burnman.AnisotropicMineral property*), 327
rho (*burnman.AnisotropicSolution property*), 525
rho (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 294
rho (*burnman.CombinedMineral property*), 266
rho (*burnman.Composite property*), 340
rho (*burnman.ElasticSolution property*), 286
rho (*burnman.Layer property*), 637
rho (*burnman.Material property*), 234
rho (*burnman.Mineral property*), 260
rho (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 778
rho (*burnman.PerplexMaterial property*), 246
rho (*burnman.Planet property*), 645
rho (*burnman.RelaxedAnisotropicSolution property*), 561
rho (*burnman.RelaxedComposite property*), 349
rho (*burnman.RelaxedSolution property*), 543
rho (*burnman.Solution property*), 277
RKprime (*class in burnman.eos.reciprocal_kprime*), 390
rotation_experiment() (*in module examples.example_hyperelastic_stress_rates*), 189
rotation_matrix (*burnman.AnisotropicMineral property*), 313
rotation_matrix (*burnman.AnisotropicSolution property*), 525
rotation_matrix (*burnman.RelaxedAnisotropicSolution property*), 561
rotation_to_crystallographic_frame() (*in module burnman.utils.unitcell*), 826
rotation_z() (*in module examples.example_hyperelastic_stress_rates*), 187
round_to_n() (*in module burnman.utils.math*), 835
row_reduce() (*in module burnman.utils.reductions*), 840
run_cli_program_with_input() (*in module burnman.utils.misc*), 845

S

S (*burnman.AnisotropicMaterial property*), 301
S (*burnman.AnisotropicMineral property*), 318
S (*burnman.AnisotropicSolution property*), 513
S (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
S (*burnman.CombinedMineral property*), 262
S (*burnman.Composite property*), 337
S (*burnman.ElasticSolution property*), 283
S (*burnman.Layer property*), 637
S (*burnman.Material property*), 234
S (*burnman.Mineral property*), 256
S (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
S (*burnman.PerplexMaterial property*), 243
S (*burnman.Planet property*), 645
S (*burnman.RelaxedAnisotropicSolution property*), 550
S (*burnman.RelaxedComposite property*), 344
S (*burnman.RelaxedSolution property*), 537
S (*burnman.Solution property*), 274
sample_path() (*in module examples.example_hyperelastic_stress_rates*), 188
Seismic1DModel (*class in burnman.classes.seismic*), 659
SeismicTable (*class in burnman.classes.seismic*), 662
set_averaging_scheme() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 294
set_averaging_scheme() (*burnman.Composite method*), 332
set_averaging_scheme() (*burnman.RelaxedComposite method*), 349
set_components() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 294
set_components() (*burnman.Composite method*), 335
set_components() (*burnman.RelaxedComposite method*), 349
set_composition() (*burnman.AnisotropicSolution method*), 512
set_composition() (*burnman.ElasticSolution method*), 279
set_composition() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 778
set_composition() (*burnman.RelaxedAnisotropicSolution method*), 546

set_composition() (*burnman.RelaxedSolution method*), 535
set_composition() (*burnman.Solution method*), 269
set_compositions_and_state_from_parameters() (*in module burnman.tools.equilibration*), 803
set_fractions() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 295
set_fractions() (*burnman.Composite method*), 332
set_fractions() (*burnman.RelaxedComposite method*), 350
set_material() (*burnman.Layer method*), 630
set_method() (*burnman.AnisotropicMaterial method*), 308
set_method() (*burnman.AnisotropicMineral method*), 327
set_method() (*burnman.AnisotropicSolution method*), 525
set_method() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 295
set_method() (*burnman.CombinedMineral method*), 266
set_method() (*burnman.Composite method*), 332
set_method() (*burnman.ElasticSolution method*), 279
set_method() (*burnman.Material method*), 223
set_method() (*burnman.Mineral method*), 248
set_method() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 778
set_method() (*burnman.PerplexMaterial method*), 246
set_method() (*burnman.RelaxedAnisotropicSolution method*), 561
set_method() (*burnman.RelaxedComposite method*), 350
set_method() (*burnman.RelaxedSolution method*), 543
set_method() (*burnman.Solution method*), 269
set_model_thermal_gradients() (*burnman.BoundaryLayerPerturbation method*), 638
set_params() (*burnman.optimize.eos_fitting.MineralFit method*), 784
set_params() (*burnman.optimize.eos_fitting.MineralFitV method*), 786
set_params() (*burnman.optimize.eos_fitting.SolutionFit method*), 788
set_params() (*burnman.optimize.nonlinear_fitting.NonLinearModel method*), 791
set_pressure_mode() (*burnman.Layer method*), 630
set_pressure_mode() (*burnman.Planet method*), 640
set_return_type() (*burnman.MaterialPolytope method*), 612
set_state() (*burnman.AnisotropicMaterial method*), 309
set_state() (*burnman.AnisotropicMineral method*), 312
set_state() (*burnman.AnisotropicSolution method*), 512
set_state() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 288
set_state() (*burnman.CombinedMineral method*), 261
set_state() (*burnman.Composite method*), 332
set_state() (*burnman.ElasticSolution method*), 280
set_state() (*burnman.Material method*), 224
set_state() (*burnman.Mineral method*), 249
set_state() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 779
set_state() (*burnman.PerplexMaterial method*), 235
set_state() (*burnman.RelaxedAnisotropicSolution method*), 545
set_state() (*burnman.RelaxedComposite method*), 342
set_state() (*burnman.RelaxedSolution method*), 534
set_state() (*burnman.Solution method*), 269
set_state_with_volume() (*burnman.AnisotropicMaterial method*), 309

set_state_with_volume() (*burnman.AnisotropicMineral method*), 327
set_state_with_volume() (*burnman.AnisotropicSolution method*), 525
set_state_with_volume() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 295
set_state_with_volume() (*burnman.CombinedMineral method*), 266
set_state_with_volume() (*burnman.Composite method*), 340
set_state_with_volume() (*burnman.ElasticSolution method*), 286
set_state_with_volume() (*burnman.Material method*), 224
set_state_with_volume() (*burnman.Mineral method*), 249
set_state_with_volume() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 779
set_state_with_volume() (*burnman.PerplexMaterial method*), 246
set_state_with_volume() (*burnman.RelaxedAnisotropicSolution method*), 546
set_state_with_volume() (*burnman.RelaxedComposite method*), 350
set_state_with_volume() (*burnman.RelaxedSolution method*), 535
set_state_with_volume() (*burnman.Solution method*), 277
set_temperature_mode() (*burnman.Layer method*), 630
setdefault() (*burnman.utils.misc.OrderedCounter method*), 841
shear_modulus (*burnman.AnisotropicMaterial property*), 309
shear_modulus (*burnman.AnisotropicMineral property*), 313
shear_modulus (*burnman.AnisotropicSolution property*), 526
shear_modulus (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 295
shear_modulus (*burnman.CombinedMineral property*), 261
shear_modulus (*burnman.Composite property*), 340
shear_modulus (*burnman.ElasticSolution property*), 281
shear_modulus (*burnman.Layer property*), 635
shear_modulus (*burnman.Material property*), 230
shear_modulus (*burnman.Mineral property*), 251
shear_modulus (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 779
shear_modulus (*burnman.PerplexMaterial property*), 238
shear_modulus (*burnman.Planet property*), 643
shear_modulus (*burnman.RelaxedAnisotropicSolution property*), 562
shear_modulus (*burnman.RelaxedComposite property*), 350
shear_modulus (*burnman.RelaxedSolution property*), 543
shear_modulus (*burnman.Solution property*), 271
shear_modulus() (*burnman.eos.aa.AA method*), 409
shear_modulus() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
shear_modulus() (*burnman.eos.birch_murnaghan.BM3 method*), 363
shear_modulus() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 368
shear_modulus() (*burnman.eos.birch_murnaghan.BM4 method*), 369
shear_modulus() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
shear_modulus() (*burnman.eos.cork.CORK method*), 416
shear_modulus() (*burnman.eos.dks_liquid.DKS_L method*), 421
shear_modulus() (*burnman.eos.dks_solid.DKS_S method*), 424
shear_modulus() (*burnman.eos.EquationOfState method*), 403
shear_modulus() (*burnman.eos.hp.HP98 method*), 428

shear_modulus() (*burnman.eos.hp.HP_TMT method*), 432
shear_modulus() (*burnman.eos.hp.HP_TMTL method*), 436
shear_modulus() (*burnman.eos.IsothermalEquationOfState method*), 358
shear_modulus() (*burnman.eos.macaw.MACAW method*), 376
shear_modulus() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
shear_modulus() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 447
shear_modulus() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
shear_modulus() (*burnman.eos.modified_tait.MT method*), 379
shear_modulus() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 454
shear_modulus() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWithAlbers method*), 459
shear_modulus() (*burnman.eos.morse_potential.Morse method*), 383
shear_modulus() (*burnman.eos.murnaghan.Murnaghan method*), 387
shear_modulus() (*burnman.eos.reciprocal_kprime.RKprime method*), 390
shear_modulus() (*burnman.eos.slb.SLB2 method*), 463
shear_modulus() (*burnman.eos.slb.SLB3 method*), 468
shear_modulus() (*burnman.eos.slb.SLB3Conductive method*), 472
shear_modulus() (*burnman.eos.slb.SLBBase method*), 472
shear_modulus() (*burnman.eos.spock.SPOCK method*), 394
shear_modulus() (*burnman.eos.vinet.Vinet method*), 398
shear_modulus() (*in module burnman.eos.morse_potential*), 483
shear_modulus() (*in module burnman.eos.reciprocal_kprime*), 489
shear_modulus_second_order() (*in module burnman.eos.birch_murnaghan*), 479
shear_modulus_third_order() (*in module burnman.eos.birch_murnaghan*), 479
shear_wave_velocity (*burnman.AnisotropicMaterial property*), 310
shear_wave_velocity (*burnman.AnisotropicMineral property*), 327
shear_wave_velocity (*burnman.AnisotropicSolution property*), 526
shear_wave_velocity (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 296
shear_wave_velocity (*burnman.CombinedMineral property*), 262
shear_wave_velocity (*burnman.Composite property*), 335
shear_wave_velocity (*burnman.ElasticSolution property*), 281
shear_wave_velocity (*burnman.Layer property*), 635
shear_wave_velocity (*burnman.Material property*), 231
shear_wave_velocity (*burnman.Mineral property*), 255
shear_wave_velocity (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 779
shear_wave_velocity (*burnman.PerplexMaterial property*), 239
shear_wave_velocity (*burnman.Planet property*), 644
shear_wave_velocity (*burnman.RelaxedAnisotropicSolution property*), 562
shear_wave_velocity (*burnman.RelaxedComposite property*), 351
shear_wave_velocity (*burnman.RelaxedSolution property*), 543
shear_wave_velocity (*burnman.Solution property*), 271
Sigmoid (*class in burnman.eos.anharmonic_prefactor_models*),

477
simplify_composite_with_composition() (*in module burnman.tools.polytope*), 824
SINGULAR_FAIL (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
SINGULAR_SUCCESS (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
site_formula() (*burnman.AnisotropicSolution method*), 526
site_formula() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 779
site_formula() (*burnman.RelaxedAnisotropicSolution method*), 562
site_formula() (*burnman.RelaxedSolution method*), 543
site_formula() (*burnman.Solution method*), 269
site_occupancies (*burnman.AnisotropicSolution property*), 526
site_occupancies (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 780
site_occupancies (*burnman.RelaxedAnisotropicSolution property*), 562
site_occupancies (*burnman.RelaxedSolution property*), 544
site_occupancies (*burnman.Solution property*), 269
site_occupancies_to_strings() (*in module burnman.utils.chemistry*), 830
SLB() (*in module burnman.eos.debye_temperature_models*), 481
SLB2 (*class in burnman.eos.slb*), 460
SLB3 (*class in burnman.eos.slb*), 463
SLB3Conductive (*class in burnman.eos.slb*), 468
SLBBase (*class in burnman.eos.slb*), 472
Slow (*class in burnman.classes.seismic*), 669
smooth_array() (*in module burnman.utils.math*), 836
SolidSolution (*in module burnman*), 279
Solution (*class in burnman*), 267
Solution (*class in burnman.optimize.nonlinear_solvers*), 799
solution_polytope_from_charge_balance() (*in module burnman.tools.polytope*), 823
solution_polytope_from_endmember_occupancies() (*in module burnman.tools.polytope*), 824
SolutionFit (*class in burnman.optimize.eos_fitting*), 788
SolutionModel (*class in burnman*), 528
solve() (*burnman.optimize.nonlinear_solvers.DampedNewtonSolve method*), 801
sort_element_list_to_IUPAC_order() (*in module burnman.utils.chemistry*), 832
sort_table() (*in module burnman.utils.misc*), 843
SPOCK (*class in burnman.eos.spock*), 393
StaceyContinental (*class in burnman.classes.geotherm*), 625
StaceyOceanic (*class in burnman.classes.geotherm*), 625
standard_psi_function() (*burnman.AnisotropicMineral method*), 312
standard_psi_function() (*burnman.AnisotropicSolution method*), 526
standard_psi_function() (*burnman.RelaxedAnisotropicSolution method*), 562
stoichiometric_array (*burnman.AnisotropicSolution property*), 526
stoichiometric_array (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 296
stoichiometric_array (*burnman.Composite property*), 336
stoichiometric_array (*burnman.ElasticSolution property*), 282
stoichiometric_array (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 780
stoichiometric_array (*burnman.RelaxedAnisotropicSolution property*), 562
stoichiometric_array (*burnman.RelaxedComposite property*), 351
stoichiometric_array (*burnman.RelaxedSolution property*), 544
stoichiometric_array (*burnman.Solution property*), 272
stoichiometric_matrix (*burnman.AnisotropicSolution property*), 526
stoichiometric_matrix (*burn-*

man.classes.mineral_helpers.HelperSpinTransition property), 296
stoichiometric_matrix (*burnman.Composite property*), 335
stoichiometric_matrix (*burnman.ElasticSolution property*), 282
stoichiometric_matrix (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 780
stoichiometric_matrix (*burnman.RelaxedAnisotropicSolution property*), 562
stoichiometric_matrix (*burnman.RelaxedComposite property*), 351
stoichiometric_matrix (*burnman.RelaxedSolution property*), 544
stoichiometric_matrix (*burnman.Solution property*), 272
store_iterates (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
STW105 (*class in burnman.classes.seismic*), 675
subpolytope_from_independent_endmember_limits() (*burnman.MaterialPolytope method*), 613
subpolytope_from_site_occupancy_limits() (*burnman.MaterialPolytope method*), 613
SubregularSolution (*class in burnman.classes.solutionmodel*), 591
subtract() (*burnman.utils.misc.OrderedCounter method*), 841
success (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
SUCCESS (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
sum_formulae() (*in module burnman.utils.chemistry*), 828
SymmetricRegularSolution (*class in burnman.classes.solutionmodel*), 584

T

T (*burnman.AnisotropicMaterial property*), 301
T (*burnman.AnisotropicMineral property*), 318
T (*burnman.AnisotropicSolution property*), 513
T (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
T (*burnman.CombinedMineral property*), 263
T (*burnman.Composite property*), 337
T (*burnman.ElasticSolution property*), 283
T (*burnman.Layer property*), 636
T (*burnman.Material property*), 233
T (*burnman.Mineral property*), 256
T (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
T (*burnman.PerplexMaterial property*), 243
T (*burnman.Planet property*), 645
T (*burnman.RelaxedAnisotropicSolution property*), 550
T (*burnman.RelaxedComposite property*), 344
T (*burnman.RelaxedSolution property*), 537
T (*burnman.Solution property*), 274
tait_constants() (*in module burnman.eos.modified_tait*), 483
temperature (*burnman.AnisotropicMaterial property*), 310
temperature (*burnman.AnisotropicMineral property*), 328
temperature (*burnman.AnisotropicSolution property*), 526
temperature (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 296
temperature (*burnman.CombinedMineral property*), 266
temperature (*burnman.Composite property*), 340
temperature (*burnman.ElasticSolution property*), 287
temperature (*burnman.Layer property*), 632
temperature (*burnman.Material property*), 226
temperature (*burnman.Mineral property*), 260
temperature (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 780
temperature (*burnman.PerplexMaterial property*), 247
temperature (*burnman.Planet property*), 641
temperature (*burnman.RelaxedAnisotropicSolution property*), 562
temperature (*burnman.RelaxedComposite property*), 351
temperature (*burnman.RelaxedSolution property*), 544
temperature (*burnman.Solution property*), 278
temperature() (*burnman.BoundaryLayerPerturbation method*), 638

temperatures() (*burnman.classes.geotherm.AdiabaticGeotherm method*), 629
temperatures() (*burnman.classes.geotherm.Anderson method*), 624
temperatures() (*burnman.classes.geotherm.Anzellini2013 method*), 627
temperatures() (*burnman.classes.geotherm.BrownShankland method*), 624
temperatures() (*burnman.classes.geotherm.Geotherm method*), 623
temperatures() (*burnman.classes.geotherm.GeothermFromPressures method*), 627
temperatures() (*burnman.classes.geotherm.Katsura2022 method*), 626
temperatures() (*burnman.classes.geotherm.Plesa2022Mars6cm3 method*), 626
temperatures() (*burnman.classes.geotherm.StaceyContinental method*), 625
temperatures() (*burnman.classes.geotherm.StaceyOceanic method*), 625
temperatures_from_pressures() (*burnman.classes.geotherm.AdiabaticGeotherm method*), 628
temperatures_from_pressures() (*burnman.classes.geotherm.Anzellini2013 method*), 628
temperatures_from_pressures() (*burnman.classes.geotherm.GeothermFromPressures method*), 627
terminate() (*burnman.optimize.nonlinear_solvers.Solution method*), 800
TerminationCode (*class in burnman.optimize.nonlinear_solvers*), 798
text (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800
thermal_energy() (*in module burnman.eos.debye*), 481
thermal_energy() (*in module burnman.eos.einstein*), 482
thermal_expansivity (*burnman.AnisotropicMaterial property*), 310
thermal_expansivity (*burnman.AnisotropicMineral property*), 328
thermal_expansivity (*burnman.AnisotropicSolution property*), 527
thermal_expansivity (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 296
thermal_expansivity (*burnman.CombinedMineral property*), 261
thermal_expansivity (*burnman.Composite property*), 333
thermal_expansivity (*burnman.ElasticSolution property*), 281
thermal_expansivity (*burnman.Layer property*), 636
thermal_expansivity (*burnman.Material property*), 232
thermal_expansivity (*burnman.Mineral property*), 251
thermal_expansivity (*burnman.optimize.composition_fitting.DummyCompositionSolut property*), 780
thermal_expansivity (*burnman.PerplexMaterial property*), 237
thermal_expansivity (*burnman.Planet property*), 644
thermal_expansivity (*burnman.RelaxedAnisotropicSolution property*), 548
thermal_expansivity (*burnman.RelaxedComposite property*), 343
thermal_expansivity (*burnman.RelaxedSolution property*), 536
thermal_expansivity (*burnman.Solution property*), 272
thermal_expansivity() (*burnman.eos.aa.AA method*), 409
thermal_expansivity() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 375
thermal_expansivity() (*burnman.eos.birch_murnaghan.BM3 method*), 363

thermal_expansivity() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 368
thermal_expansivity() (*burnman.eos.birch_murnaghan.BM4 method*), 371
thermal_expansivity() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
thermal_expansivity() (*burnman.eos.cork.CORK method*), 416
thermal_expansivity() (*burnman.eos.dks_liquid.DKS_L method*), 421
thermal_expansivity() (*burnman.eos.dks_solid.DKS_S method*), 425
thermal_expansivity() (*burnman.eos.EquationOfState method*), 404
thermal_expansivity() (*burnman.eos.hp.HP98 method*), 428
thermal_expansivity() (*burnman.eos.hp.HP_TMT method*), 432
thermal_expansivity() (*burnman.eos.hp.HP_TMTL method*), 436
thermal_expansivity() (*burnman.eos.IsothermalEquationOfState method*), 355
thermal_expansivity() (*burnman.eos.macaw.MACAW method*), 379
thermal_expansivity() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
thermal_expansivity() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 447
thermal_expansivity() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
thermal_expansivity() (*burnman.eos.modified_tait.MT method*), 382
thermal_expansivity() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
thermal_expansivity() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWithA method*), 457
thermal_expansivity() (*burnman.eos.morse_potential.Morse method*), 386
thermal_expansivity() (*burnman.eos.murnaghan.Murnaghan method*), 390
thermal_expansivity() (*burnman.eos.reciprocal_kprime.RKprime method*), 393
thermal_expansivity() (*burnman.eos.slb.SLB2 method*), 463
thermal_expansivity() (*burnman.eos.slb.SLB3 method*), 468
thermal_expansivity() (*burnman.eos.slb.SLB3Conductive method*), 472
thermal_expansivity() (*burnman.eos.slb.SLBBase method*), 472
thermal_expansivity() (*burnman.eos.spock.SPOCK method*), 397
thermal_expansivity() (*burnman.eos.vinet.Vinet method*), 401
thermal_expansivity_tensor (*burnman.AnisotropicMineral property*), 316
thermal_expansivity_tensor (*burnman.AnisotropicSolution property*), 527
thermal_expansivity_tensor (*burnman.RelaxedAnisotropicSolution property*), 547
thermal_stress_tensor (*burnman.AnisotropicMineral property*), 328
thermal_stress_tensor (*burnman.AnisotropicSolution property*), 527
thermal_stress_tensor (*burnman.RelaxedAnisotropicSolution property*), 547
to_string() (*burnman.AnisotropicMaterial method*), 310
to_string() (*burnman.AnisotropicMineral method*), 328
to_string() (*burnman.AnisotropicSolution method*), 527
to_string() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 296
to_string() (*burnman.CombinedMineral method*), 267
to_string() (*burnman.Composite method*), 333

to_string() (*burnman.ElasticSolution method*), 287
to_string() (*burnman.Material method*), 223
to_string() (*burnman.Mineral method*), 248
to_string() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 780
to_string() (*burnman.PerplexMaterial method*), 247
to_string() (*burnman.RelaxedAnisotropicSolution method*), 563
to_string() (*burnman.RelaxedComposite method*), 351
to_string() (*burnman.RelaxedSolution method*), 544
to_string() (*burnman.Solution method*), 278
TOO_NONLINEAR (*burnman.optimize.nonlinear_solvers.TerminationCode attribute*), 799
total() (*burnman.utils.misc.OrderedCounter method*), 842
transform_solution_to_new_basis() (*in module burnman.tools.solution*), 815
tvel_formatted_data_and_header() (*in module burnman.tools.output_seismo*), 822

U

unit_normalize() (*in module burnman.utils.math*), 835
unroll() (*burnman.AnisotropicMaterial method*), 311
unroll() (*burnman.AnisotropicMineral method*), 329
unroll() (*burnman.AnisotropicSolution method*), 527
unroll() (*burnman.classes.mineral_helpers.HelperSpinTransition method*), 296
unroll() (*burnman.CombinedMineral method*), 267
unroll() (*burnman.Composite method*), 332
unroll() (*burnman.ElasticSolution method*), 287
unroll() (*burnman.Material method*), 225
unroll() (*burnman.Mineral method*), 248
unroll() (*burnman.optimize.composition_fitting.DummyCompositionSolution method*), 780
unroll() (*burnman.PerplexMaterial method*), 247
unroll() (*burnman.RelaxedAnisotropicSolution method*), 563
unroll() (*burnman.RelaxedComposite method*), 351
unroll() (*burnman.RelaxedSolution method*), 545
unroll() (*burnman.Solution method*), 278
unrotated_cell_vectors (*burnman.AnisotropicMineral property*), 312
unrotated_cell_vectors (*burnman.AnisotropicSolution property*), 527
unrotated_cell_vectors (*burnman.RelaxedAnisotropicSolution property*), 563
update() (*burnman.utils.misc.OrderedCounter method*), 842

V

V (*burnman.AnisotropicMaterial property*), 301
V (*burnman.AnisotropicMineral property*), 318
V (*burnman.AnisotropicSolution property*), 513
V (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 288
V (*burnman.CombinedMineral property*), 263
V (*burnman.Composite property*), 337
V (*burnman.ElasticSolution property*), 283
V (*burnman.Layer property*), 637
V (*burnman.Material property*), 234
V (*burnman.Mineral property*), 256
V (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 772
V (*burnman.PerplexMaterial property*), 243
V (*burnman.Planet property*), 645
V (*burnman.RelaxedAnisotropicSolution property*), 550
V (*burnman.RelaxedComposite property*), 344
V (*burnman.RelaxedSolution property*), 537
V (*burnman.Solution property*), 274
v_p (*burnman.AnisotropicMaterial property*), 311
v_p (*burnman.AnisotropicMineral property*), 329
v_p (*burnman.AnisotropicSolution property*), 528
v_p (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 297
v_p (*burnman.CombinedMineral property*), 267
v_p (*burnman.Composite property*), 341
v_p (*burnman.ElasticSolution property*), 287
v_p (*burnman.Layer property*), 637
v_p (*burnman.Material property*), 234
v_p (*burnman.Mineral property*), 260

v_p (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 781
v_p (*burnman.PerplexMaterial property*), 247
v_p (*burnman.Planet property*), 646
v_p (*burnman.RelaxedAnisotropicSolution property*), 563
v_p (*burnman.RelaxedComposite property*), 352
v_p (*burnman.RelaxedSolution property*), 545
v_p (*burnman.Solution property*), 278
v_p() (*burnman.classes.seismic.AK135 method*), 684
v_p() (*burnman.classes.seismic.Fast method*), 674
v_p() (*burnman.classes.seismic.IASP91 method*), 681
v_p() (*burnman.classes.seismic.PREM method*), 668
v_p() (*burnman.classes.seismic.Seismic1DModel method*), 660
v_p() (*burnman.classes.seismic.SeismicTable method*), 663
v_p() (*burnman.classes.seismic.Slow method*), 671
v_p() (*burnman.classes.seismic.STW105 method*), 677
v_phi (*burnman.AnisotropicMaterial property*), 311
v_phi (*burnman.AnisotropicMineral property*), 329
v_phi (*burnman.AnisotropicSolution property*), 528
v_phi (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 297
v_phi (*burnman.CombinedMineral property*), 267
v_phi (*burnman.Composite property*), 341
v_phi (*burnman.ElasticSolution property*), 287
v_phi (*burnman.Layer property*), 637
v_phi (*burnman.Material property*), 234
v_phi (*burnman.Mineral property*), 260
v_phi (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 781
v_phi (*burnman.PerplexMaterial property*), 247
v_phi (*burnman.Planet property*), 646
v_phi (*burnman.RelaxedAnisotropicSolution property*), 563
v_phi (*burnman.RelaxedComposite property*), 352
v_phi (*burnman.RelaxedSolution property*), 545
v_phi (*burnman.Solution property*), 278
v_phi() (*burnman.classes.seismic.AK135 method*), 684
v_phi() (*burnman.classes.seismic.Fast method*), 675
v_phi() (*burnman.classes.seismic.IASP91 method*), 681
v_phi() (*burnman.classes.seismic.PREM method*), 668
v_phi() (*burnman.classes.seismic.Seismic1DModel method*), 660
v_phi() (*burnman.classes.seismic.SeismicTable method*), 665
v_phi() (*burnman.classes.seismic.Slow method*), 671
v_phi() (*burnman.classes.seismic.STW105 method*), 678
v_s (*burnman.AnisotropicMaterial property*), 311
v_s (*burnman.AnisotropicMineral property*), 329
v_s (*burnman.AnisotropicSolution property*), 528
v_s (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 297
v_s (*burnman.CombinedMineral property*), 267
v_s (*burnman.Composite property*), 341
v_s (*burnman.ElasticSolution property*), 287
v_s (*burnman.Layer property*), 637
v_s (*burnman.Material property*), 234
v_s (*burnman.Mineral property*), 260
v_s (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 781
v_s (*burnman.PerplexMaterial property*), 247
v_s (*burnman.Planet property*), 646
v_s (*burnman.RelaxedAnisotropicSolution property*), 563
v_s (*burnman.RelaxedComposite property*), 352
v_s (*burnman.RelaxedSolution property*), 545
v_s (*burnman.Solution property*), 278
v_s() (*burnman.classes.seismic.AK135 method*), 684
v_s() (*burnman.classes.seismic.Fast method*), 675
v_s() (*burnman.classes.seismic.IASP91 method*), 681
v_s() (*burnman.classes.seismic.PREM method*), 668
v_s() (*burnman.classes.seismic.Seismic1DModel method*), 660
v_s() (*burnman.classes.seismic.SeismicTable method*), 663
v_s() (*burnman.classes.seismic.Slow method*), 672
v_s() (*burnman.classes.seismic.STW105 method*), 678
validate() (*burnman.optimize.eos_fitting.MineralFit*

method), 784
validate() (*burnman.optimize.eos_fitting.MineralFitV method*), 787
validate() (*burnman.optimize.eos_fitting.SolutionFit method*), 788
validate() (*burnman.optimize.nonlinear_fitting.NonLinearModel method*), 792
validate_parameters() (*burnman.eos.aa.AA method*), 409
validate_parameters() (*burnman.eos.anharmonic_debye.AnharmonicDebye static method*), 476
validate_parameters() (*burnman.eos.anharmonic_prefactor_models.AnharmonicPrefactorModel method*), 476
validate_parameters() (*burnman.eos.anharmonic_prefactor_models.PowerLaw method*), 477
validate_parameters() (*burnman.eos.anharmonic_prefactor_models.Sigmoid method*), 477
validate_parameters() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
validate_parameters() (*burnman.eos.birch_murnaghan.BM3 method*), 364
validate_parameters() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 368
validate_parameters() (*burnman.eos.birch_murnaghan.BM4 method*), 369
validate_parameters() (*burnman.eos.brosh_calphad.BroshCalphad method*), 413
validate_parameters() (*burnman.eos.cork.CORK method*), 417
validate_parameters() (*burnman.eos.dks_liquid.DKS_L method*), 421
validate_parameters() (*burnman.eos.dks_solid.DKS_S method*), 425
validate_parameters() (*burnman.eos.EquationOfState method*), 406
validate_parameters() (*burnman.eos.hp.HP98 method*), 428
validate_parameters() (*burnman.eos.hp.HP_TMT method*), 433
validate_parameters() (*burnman.eos.hp.HP_TMTL method*), 436
validate_parameters() (*burnman.eos.IsothermalEquationOfState method*), 359
validate_parameters() (*burnman.eos.macaw.MACAW method*), 376
validate_parameters() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
validate_parameters() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 447
validate_parameters() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 448
validate_parameters() (*burnman.eos.modified_tait.MT method*), 380
validate_parameters() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGD method*), 452
validate_parameters() (*burnman.eos.modular_mie_grueneisen_debye.ModularMGDWit method*), 457
validate_parameters() (*burnman.eos.morse_potential.Morse method*), 383
validate_parameters() (*burnman.eos.murnaghan.Murnaghan method*), 387
validate_parameters() (*burnman.eos.reciprocal_kprime.RKprime method*), 391
validate_parameters() (*burnman.eos.slb.SLB2 method*), 463
validate_parameters() (*burnman.eos.slb.SLB3 method*), 468
validate_parameters() (*burnman.eos.slb.SLB3Conductive method*), 472
validate_parameters() (*burnman.eos.slb.SLBBase method*), 473

validate_parameters() (*burnman.eos.spock.SPOCK method*), 394
validate_parameters() (*burnman.eos.vinet.Vinet method*), 398
value() (*burnman.eos.anharmonic_prefactor_models.AnharmonicPrefactorModel method*), 476
value() (*burnman.eos.anharmonic_prefactor_models.PowerLaw method*), 476
value() (*burnman.eos.anharmonic_prefactor_models.Sigmoid method*), 477
value() (*burnman.eos.debye_temperature_models.DebyeTemperatureModel method*), 478
values() (*burnman.utils.misc.OrderedCounter method*), 842
Vinet (*class in burnman.eos.vinet*), 397
visual_center_of_polygon() (*in module burnman.utils.plotting*), 835
Voigt (*class in burnman.averaging_schemes*), 617
voigt_array_from_cijs() (*in module burnman.utils.anisotropy*), 833
voigt_index_to_ij() (*in module burnman.utils.anisotropy*), 833
voigt_notation_to_compliance_tensor() (*in module burnman.utils.anisotropy*), 833
voigt_notation_to_stiffness_tensor() (*in module burnman.utils.anisotropy*), 833
VoigtReussHill (*class in burnman.averaging_schemes*), 619
volume (*burnman.AnisotropicMaterial property*), 311
volume (*burnman.AnisotropicMineral property*), 329
volume (*burnman.AnisotropicSolution property*), 528
volume (*burnman.classes.mineral_helpers.HelperSpinTransition property*), 297
volume (*burnman.CombinedMineral property*), 267
volume (*burnman.Composite property*), 341
volume (*burnman.ElasticSolution property*), 287
volume (*burnman.Material property*), 233
volume (*burnman.Mineral property*), 260
volume (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 781
volume (*burnman.PerplexMaterial property*), 248
volume (*burnman.RelaxedAnisotropicSolution property*), 564
volume (*burnman.RelaxedComposite property*), 352
volume (*burnman.RelaxedSolution property*), 545
volume (*burnman.Solution property*), 279
volume() (*burnman.Calibrant method*), 353
volume() (*burnman.eos.aa.AA method*), 408
volume() (*burnman.eos.birch_murnaghan.BirchMurnaghanBase method*), 372
volume() (*burnman.eos.birch_murnaghan.BM3 method*), 364
volume() (*burnman.eos.birch_murnaghan.BM3Shear2 method*), 368
volume() (*burnman.eos.birch_murnaghan.BM4 method*), 348
volume() (*burnman.eos.brosh_calphad.BroshCalphad method*), 412
volume() (*burnman.eos.cork.CORK method*), 416
volume() (*burnman.eos.dks_liquid.DKS_L method*), 420
volume() (*burnman.eos.dks_solid.DKS_S method*), 424
volume() (*burnman.eos.EquationOfState method*), 401
volume() (*burnman.eos.hp.HP98 method*), 428
volume() (*burnman.eos.hp.HP_TMT method*), 432
volume() (*burnman.eos.hp.HP_TMTL method*), 436
volume() (*burnman.eos.IsothermalEquationOfState method*), 359
volume() (*burnman.eos.macaw.MACAW method*), 376
volume() (*burnman.eos.mie_grueneisen_debye.MGD2 method*), 442
volume() (*burnman.eos.mie_grueneisen_debye.MGD3 method*), 447
volume() (*burnman.eos.mie_grueneisen_debye.MGDBase method*), 447
volume() (*burnman.eos.modified_tait.MT method*), 379
volume() (*burnman.eos.modular_mie_grueneisen_debye.ModularM method*), 451
volume() (*burnman.eos.modular_mie_grueneisen_debye.ModularM method*), 460
volume() (*burnman.eos.morse_potential.Morse method*), 383
volume() (*burnman.eos.murnaghan.Murnaghan method*), 386
volume() (*burnman.eos.reciprocal_kprime.RKprime method*), 390
volume() (*burnman.eos.slb.SLB2 method*), 463
volume() (*burnman.eos.slb.SLB3 method*), 468

volume() (*burnman.eos.slb.SLB3Conductive method*), 472
volume() (*burnman.eos.slb.SLBBase method*), 472
volume() (*burnman.eos.spock.SPOCK method*), 394
volume() (*burnman.eos.vinet.Vinet method*), 397
volume() (*in module burnman.eos.modified_tait*), 483
volume() (*in module burnman.eos.morse_potential*), 483
volume() (*in module burnman.eos.murnaghan*), 483
volume_fourth_order() (*in module burnman.eos.birch_murnaghan*), 480
volume_hessian (*burnman.AnisotropicSolution property*), 528
volume_hessian (*burnman.optimize.composition_fitting.DummyCompositionSolution property*), 781
volume_hessian (*burnman.RelaxedAnisotropicSolution property*), 564
volume_hessian (*burnman.RelaxedSolution property*), 545
volume_hessian (*burnman.Solution property*), 270
volume_hessian() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
volume_hessian() (*burnman.classes.solutionmodel.FunctionSolution method*), 601
volume_hessian() (*burnman.classes.solutionmodel.IdealSolution method*), 571
volume_hessian() (*burnman.classes.solutionmodel.SubregularSolution method*), 593
volume_hessian() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 589
volume_third_order() (*in module burnman.eos.birch_murnaghan*), 480
volume_vinet() (*in module burnman.eos.vinet*), 490
VoverKT_excess() (*burnman.classes.solutionmodel.AsymmetricRegularSolution method*), 579
VoverKT_excess() (*burnman.classes.solutionmodel.FunctionSolution method*), 601
VoverKT_excess() (*burnman.classes.solutionmodel.IdealSolution method*), 572
VoverKT_excess() (*burnman.classes.solutionmodel.MechanicalSolution method*), 567
VoverKT_excess() (*burnman.classes.solutionmodel.SubregularSolution method*), 594
VoverKT_excess() (*burnman.classes.solutionmodel.SymmetricRegularSolution method*), 585
VoverKT_excess() (*burnman.SolutionModel method*), 532

W

wave_velocities() (*burnman.AnisotropicMaterial method*), 300
wave_velocities() (*burnman.AnisotropicMineral method*), 329
wave_velocities() (*burnman.AnisotropicSolution method*), 528
wave_velocities() (*burnman.RelaxedAnisotropicSolution method*), 564
weight_composition (*burnman.Composition property*), 609
weighted_constrained_least_squares() (*in module burnman.optimize.linear_fitting*), 790
weighted_residual_plot() (*in module burnman.optimize.nonlinear_fitting*), 797
write_axisem_input() (*in module burnman.tools.output_seismo*), 821
write_mineos_input() (*in module burnman.tools.output_seismo*), 822
write_tvel_file() (*in module burnman.tools.output_seismo*), 821

X

x (*burnman.optimize.nonlinear_solvers.Iterates attribute*), 798
x (*burnman.optimize.nonlinear_solvers.Solution attribute*), 800