Otros títulos

Decálogo de impactos ambientales. Geografía de las transformaciones en sistemas acuáticos de Colombia
Jhon Charles Donato-Rondón

Ecuaciones diferenciales ordinarias. Técnicas de resolución
Luz Marina Moya
Edixon Rojas

Estadística descriptiva multivariada
Campo Elías Pardo

Colección
textos

This book is an introduction to the algorithms used on geological phase equilibria modeling following the current trend in the petrological community to estimate P - T conditions for the formation and evolution of rocks. The text is intended for gaining understanding of thermodynamic modeling applied to petrology, especially in the estimation of equilibrium conditions for rocks in the lithosphere. The approach used within the book follows the application of phase equilibria concepts to some practical aspects of modeling. The book starts with a brief introduction to Python programming and the bases of linear algebra, then, it introduces the thermodynamic theoretical framework along with the basis of phase equilibria. The last two chapters present details of thermobarometric calculations and phase diagram modeling.

Geociencias

Pythonic Phase Equilibria Modeling in Petrology
Carlos A. Zuluaga C.

Pythonic phase equilibria modeling in petrology

Carlos A. Zuluaga C.

Facultad de Ciencias
Sede Bogotá

UNIVERSIDAD NACIONAL DE COLOMBIA

Germán A. Prieto
Carlos Zuluaga obtained his Ph.D at the Univeristy of Alabama (USA), after his graduation he worked as visiting assistant professor at the University of Minnesota (Morris, MN, USA) for one year and at Pomona College (Claremont, CA, USA) as assistant professor for another year. He started working at the Universidad Nacional de Colombia in 2006 where he is a full professor with tenure. Carlos teaches courses related to mapping, hard-rock geology, and tectonics. The list of classes include: mineralogy, petrology, geochemistry, thermodynamics, and geodynamics. His research interests in the last years has been the application of thermodynamic models to unravel metamorphic and tectonic histories of high-grade metamorphic rocks. He is interested in the physical and chemical properties of minerals and relations of these properties with mineral stability under various system conditions. Regionally, he is interested in understanding the tectonic evolution of the Colombian Caribbean Margin and the northern Andes of Colombia.

Pythonic phase equilibria modeling in petrology

Pythonic phase equilibria modeling in petrology

Carlos Augusto Zuluaga Castrillón

UNIVERSIDAD
NACIONAL
DE COLOMBIA

Bogotá, D. C., Colombia, Mayo de 2025

Primera edición, 2025.

ISBN 978-958-505-952-8 (papel)
ISBN 978-958-505-953-5 (digital)

Edición
Coordinación de Publicaciones Facultad de Ciencias
coorpub_fcbog@unal.edu.co

Corrección de estilo
Rosa González

Diseño de carátula
Damián Crofort

Maqueta LaTeX
Camilo Cubides

Impreso y hecho en Bogotá, D. C., Colombia.

Catalogación en la publicación Universidad Nacional de Colombia

Zuluaga Castrillón, Carlos Augusto, 1967-
Pythonic phase equilibria modeling in petrology / Carlos A. Zuluaga C. -- Primera edición. -- Bogotá : Universidad Nacional de Colombia. Facultad de Ciencias : Coordinación de publicaciones Facultad de Ciencias, 2025.
309 páginas : ilustraciones a color, diagramas. -- (Colección Textos. Geociencias)

Incluye referencias bibliográficas e índice analítico
ISBN 978-958-505-952-8 (impreso). -- ISBN 978-958-505-953-5 (digital)

1. Termodinámica -- Simulación por computadora 2. Equilibrio termodinámico -- Modelos matemáticos 3. Ley de las fases y equilibrio -- Ejercicios y problemas 4. Diagramas de fase -- Simulación por computadora 5. Diagramas de equilibrio 6. Termobarometría 7. Petrología -- Métodos de simulación 8. Modelos en geoquímica -- Problemas, ejercicios, etc. 9. Modelos en geología 10. Geología -- Matemáticas 11. Python (Lenguaje de programación) 12. Lenguajes de modelado (Computación) I. Título II. Serie

CDD-23 536.70113 / 2025

Contents

List of Figures

List of Tables

Acknowledgements

Many chapters and chapter sections of this book are based on lecture notes I developed over the years while teaching graduate courses at the Universidad Nacional de Colombia. I decided to write these lecture notes using *Jupyter notebooks* and *Python* due to the possibilities to mix code with formatted text in the notebooks, to the many advantages of using the *Python* ecosystem, and to the option of generating the book from the notebooks using the *Jupyter Book* package. I am deeply grateful to the many developers working on those open-source software tools and the people who provide online documentation and support to software development applications. I'm especially grateful for the *Python* language and its creator, Guido van Rossum.

I assembled the lecture notes and wrote some additional material (particularly Chapters 6 and 7) during my sabbatical leave in the year 2023. I extend my gratitude to the Universidad Nacional de Colombia for supporting my sabbatical leave request; thanks also to the university's Facultad de Ciencias for providing financial support for the book editorial process. A significant portion of the writing took place while visiting the University of Grenoble Alps (UGA), which provided partial financial support for the visit. I am grateful to Matthias Bernet for facilitating this visit and for reviewing an earlier version of the book. During my visit to UGA, Matthias was not only a colleague whose geological discussions prompt me to reconsider my previous understanding of several geological topics but also a friend with whom I shared holiday trips and evening dinners with his wife, Liliana.

Finally, I would like to thank my wife, Edna, and my son, Jacobo, to whom I dedicate this book.

Preface

The current trend in the petrological community to estimate $P-T$ conditions for the formation and evolution of rocks is based on phase equilibria modeling. At the time of writing this book, there are approximately five modeling programs, each following a set of strategies that users often do not fully understand. Most users have some knowledge of the laws of thermodynamics but lack a deep understanding of how to apply this knowledge to phase equilibria modeling. This book attempts to cover some of the algorithms used in the geological phase equilibria modeling programs and provides insights into the inner workings of these programs, aiming to facilitate the process of interpretation with scientific rigor.

The text was originally intended to aid in understanding thermodynamic modeling as applied to metamorphic petrology, particularly estimating equilibrium conditions for rocks in the lithosphere. However, many topics are equally applicable to phase equilibria in igneous petrology. I have not attempted to cover the entire theoretical framework or delve deeply into the subjects addressed in this book. Instead, the book intents to show some practical aspects of phase equilibria modeling as used in current modeling programs available to the petrological community. This book serves as a companion to traditional thermodynamic and phase equilibria texts.

Each chapter has a set of worked examples, some of which are challenging due to their use of complex algorithms. I have tried, wherever possible, to follow the easiest route to get to the solution; however, I may have missed a simpler solution than the one developed in the exercise. Most worked examples include complete code listings, some build on results from previous exercises, and a few have indicated steps rather than detailed lines of code to reach the final solution. I have tried to reproduce the exercises several times to ensure any reader has all the necessary information to arrive at the correct solutions.

Modeling

As this book focuses primarily on thermodynamic modeling, a discussion about models in general—and geological models in particular—is appropriate. Models are explicitly or implicitly used in geological sciences, as recognized by many geologists. For example, Krumbein (1962) stated:

> A third problem raised by quantification relates to the selection or design of models appropriate to specific geological problems. Models—in the sense of devices for organizing data—have long been used in geology, and their use is implied in Chamberlain's principle of multiple working hypothesis. (p. 1088)

In a general way, we can think of models as representations that help us understand problem structures and guide further studies; however, they are not susceptible to definitive proof. Models are useful in searching for evidence to corroborate hypothesis, to elucidate discrepancies, and to elaborate sensitivity analysis (Oreskes *et al.*, 1994). Modeling is the simulation of systems or phenomena within systems, which are generally visualized in a simplified way. In one type of modeling, the final state of a system is known, and previous states are investigated by simulation; this methodology is commonly referred to as backward modeling. Backward modeling involves solving inverse problems, which entails determining the structure of systems from samples of their response to stimuli. In other words, it requires measuring the distribution of the dependent variables in the system and from them estimating the nature of the independent variables. A second type of modeling assumes initial system conditions, and simulation reproduces one or more successive states leading to more evolved conditions. This type of modeling is commonly known as forward modeling. This book utilizes various applications of both modeling types: pseudosection construction (forward modeling) and thermobarometry (backward modeling).

Errors and Model Evaluation

Carter (2004) identified two sources of errors: measurement errors and modeling errors. Modeling inherently involves working with data containing uncertainties from measurement errors. Simplifying assumptions and numerical schemes used to solve equations in a model produce modeling

errors. Uncertainties originating from the adopted models can limit the interpretations of modeling results and impact the predictive reliability of the model.

Flawed models with a quantified uncertainty may appear reliable and credible, potentially leading to incorrect conclusions. Two examples can illustrate this issue. 1) Consider an isotopic age in a composite mineral grain with multiple origins (e.g., multiple inheritance in zircon), the resulting age reported with its uncertainty can give the impression of a precise age. However, the multiple origins render the reported age meaningless. 2) In thermobarometry, the first requirement for pressure and temperature determinations using several mineral phases in a rock is that the phases must be in equilibrium. A report presenting thermobarometric results and their uncertainties from mineral phases not in equilibrium gives the impression of reliability, but the results are, again, meaningless. Ultimately, measurement and modeling errors will correlate to how well the predictions compare to the real system and then with the uncertainty associated with the prediction.

Oreskes (1998) categorized model uncertainties (e.g., for environmental lead) as theoretical, empirical, parametrical, and temporal. Aspects of systems that are not fully understood are categorized as theoretical uncertainties. Examples of these in thermodynamic systems are distribution coefficients and molecular interactions. Aspects of systems that are difficult—or sometimes impossible—to measure are empirical uncertainties; precise atomic distribution in a crystal structure is an example of this type of uncertainty. Sampling bias and analytical uncertainty also fall within this category. Simplifying complexities of a system to provide manageable model inputs gives rise to parametrical uncertainties. For instance, macroscopic representations of activity models that rely on microscopic parameters are an example of this type of uncertainty. Temporal uncertainties arise from the assumption that systems are temporally stable.

Uncertainties in calculations are propagated using the error propagation equation:

$$\mathbf{V_y} = \mathbf{J}\mathbf{V_x}\mathbf{J}^{\mathrm{T}} \tag{1}$$

where $\mathbf{V_x}$ and $\mathbf{V_y}$ are the covariance matrices for x and y, respectively, while $\mathbf{J}$ is the Jacobian matrix, which consists of the first derivatives of the elements of y with respect to the elements of x. Measurement errors and modeling errors are considered in the uncertainty propagation. Uncertainties in the internally consistent thermodynamic dataset and uncertainties related to activity models are addressed in Chapter 7. Additionally, uncertainties in the

analytical data (e.g., oxide compositions) are considered by propagating errors related to the analytical precision within compositional vectors.

The debate over the applicability of model validation has encouraged natural scientists toward a more open approach: model evaluation. Unlike validation, which implies legitimization and assumes the model is inherently aligned with the truth, the evaluation approach allows for both positive and negative outcomes during the assessment process. According to Hodges and Dewar (1992), validatable models must meet four criteria for the situation being modeled: (a) it must be observable and measurable; (b) it must exhibit constancy of structure in time; (c) it must exhibit constancy across variations in conditions not specified in the model; and (d) it must permit the collection of ample data. Oreskes (1998) summarized those criteria as: measurability, accessibility, and temporal and spatial invariance. However, models in the natural sciences (e.g., geology) generally fail to meet these criteria and thus cannot be validated. Their reliability as a basis for prediction cannot be demonstrated because natural science deals with complex systems involving multiple interacting variables that are difficult to access, measure, or control and may change over space or time (Oreskes, 1998).

In natural systems, the process of model assessment is complicated by indeterminate or as-yet-undetermined interrelationships between variables. Despite these challenges, there are several parameters that can be used to assess a model's reliability in terms of the current knowledge of a specific discipline. First, theories can be probabilistically tested, meaning that the more experimental tests a model has successfully passed—often correlated with the length of time the model has been in use—the more likely it is to be accurate. Second, we can evaluate the quality of models based on their underlying scientific principles, the quantity and quality of input parameters, and their ability to reproduce independent empirical data.

Phase Equilibria Modeling

Geochemical modeling involves quantitative methods to evaluate large-scale or complex geochemical processes, such as mass balance and equilibrium conditions. A subset of geochemical modeling is thermodynamic computations, where the goal is to identify the most stable state of a system (equilibrium conditions) given a set of system constraints. Depending on the defined (fixed) variables and the computed variables, backward or forward thermodynamic models can be constructed. Thermobarometry is an example of backward thermodynamic modeling. Here, the goal is to determine

the pressure and temperature (independent variables, unknown) that drove the system toward its chemical equilibrium state (dependent variables, known or measured) at peak metamorphic conditions (Chapter 7). Conversely, forward thermodynamic modeling is used in the production of phase diagrams (Chapter 6). In this approach, the independent variables (pressure and temperature) are manipulated in the simulation, causing a perturbation in the system in order to study the resulting mineral responses.

Geochemical models require thermodynamic and kinetic data, which are often incompletely or only approximately known. Therefore, geochemical models are generally ill-posed problems that require additional information to obtain solutions. For thermodynamic data, we rely on additional information from experiments to constrain the problem (e.g., enthalpy determinations). The process of tuning the model (calibration) involves the manipulation of the independent variables to obtain a match between the observed and simulated distributions of dependent variables (Oreskes *et al.*, 1994).

The reproduction of thermodynamic properties using empirical formulations is improved through the use of self-consistent data usually fitted with mathematical programming techniques and regression analysis (Chapter 2). Internally consistent thermodynamic databases are used for thermodynamic analysis of geologically relevant phase diagrams using computer programs. These programs use the internally consistent databases to generate phase diagrams involving solid solutions (forward modeling) defined through activity models for the phases. The activity models include information about ideal activities, activity coefficients, and types of non-ideal interactions (ideal, symmetrical, and asymmetrical); these topics are covered in Chapter 5.

The topic of thermobarometric calculations (backward modeling) is developed in Chapter 7. Two primary approaches are described. The first is by intersection of a set of two linearized mineral reactions at or close to the pressure and temperature of equilibration (for example, the garnet-biotite thermometer and the grossular-kyanite/sillimanite-anorthite barometer). A second approach is by searching an optimal $P - T$ (pressure-temperature) point at the intersections of multiple mineral reactions.

Organization of Chapters

Chapter 1 presents a very brief introduction to *Python* programming and the basics of linear algebra used in the development of the book. The

thermodynamic theoretical framework begins with the four laws of thermodynamics, introduced in Chapter 2. Chapter 3 focuses on the geometrical analysis of compositional and reactive spaces, which, along with thermodynamics, is the basis for understanding phase equilibria. Chapters 4 and 5 address the calculations of Gibbs free energy in systems with pure phases and solid solutions. Chapters 6 and 7 form the core of the book, showcasing computational phase equilibria modeling strategies for performing thermobarometric and phase diagram calculations.

Chapter 1

Introduction

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

$Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x$

$x = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad Ax = \lambda x \quad A$

1.1. Programming Tool: *Python*

A large portion of this book is devoted to encouraging individual work by way of replication of practical programming exercises. There are many programming languages, ranging from low-level languages that use binary representations (zeros and ones) to high-level languages that resemble the way we speak. This book makes extensive use of algorithms written in the *Python* programming language as a tool for understanding and visualizing problems related to thermodynamics and phase equilibrium. Note that coded instructions in *Python* are very similar to MATLAB instructions (pre-programmed functions in *Python* follow similar calling conventions).

Python is a high-level language that is easy to learn. In particular, this book uses the *Jupyter* interface which is based on building *Notebooks*. It is recommended to go through the exercises using this interface. These are some of the advantages of using *Python*:

- *Python* is a multi-paradigm programming language offering several options for software engineering: imperative programming, functional programming, and object-oriented programming (OOP); thus, it does not force any particular style of programming. OOP is based on the idea of using objects instead of procedures as in traditional programming. Objects are created (instantiated) based on components of the program termed *classes*, which are defined by their properties (constants and variables) and methods (class functions).
- *Python* is freely distributed (BSD license) and works on all operating systems (Windows, macOS, Linux, etc.).
- *Python* is based on an interactive interpreter (It is a language that allows interactive experimentation, enabling users to test and refine their code directly).
- *Python* has a simple syntax, making it easy to learn, read, and understand. It is supported by comprehensive documentation.
- *Python* provides a rich set of built-in data types.
- *Python* includes an extensive standard library for various applications.
- *Python* supports a wide range of mature external libraries (packages) (*NumPy, SciPy, Matplotlib*, etc.)
- Many *Python* libraries are adapted or imported from routines extensively tested in other languages (C, C++, and Fortran).

As you learn *Python* (if you haven't already), you will notice some differences with other programming languages. In particular, it is important to bear in mind that the content of code blocks (*loops, functions, classes,* etc.) is delimited by indentation—spaces or tabs before each command line that belongs to the block—. Compare this to other programming languages that declare blocks using a set of characters (e.g., braces *{}*). Both spaces and tabs can be used to indent code; however, it is recommended not to mix them. The *Python* community strongly encourages programmers to follow the *Style Guide for Python Code*, also known as *Python Enhancement Proposal #8* or PEP 8 (Rossum *et al.*, 2001). Some of the recommendations are: (i) use of spaces instead of tabs for indentation, (ii) use of four spaces for each level of indentation, (iii) keep lines to 79 characters or less, (iv) avoid spaces around list or array indices, and (v) use a single space before and after the equal sign ("=") in variable assignments (e.g., write x = 10, not x= 10 or x=10).

1.2. Jupyter Notebooks

Notebooks are documents that combine text and fragments of programs that can be executed interactively. This enables the development of programs alongside explanatory notes or documents (e.g., articles) that include the necessary code to reproduce results such as data tables and figures. Notebooks can also be exported to different formats to present the final version of the document (e.g., .pdf).

1.2.1. The Cells of the Notebook

Notebooks are composed of cells, which may contain code (program execution instructions), text, or both. The code in the cells can be executed by selecting the cell and using the appropriate menu button or by using the "Shift" + "Enter" key combination. When cells with code are executed, a new cell is produced with the result of the execution of the program. When cells with text that has formatting (`Markdown`) are executed, the text is displayed as it would finally appear in a PDF or Word file.

1.2.1.1. Formatted Text

Cells with text can be formatted using a language called `Markdown`. Using this type of cell in combination with cells with code, a self-contained complete document with executable scripts can be created. Within `Markdown`

cells, symbols and mathematical formulas can be included in the text using the symbol $ at the start and end of the block, once for on-line use within the text and twice ($$) for equations on their own line. Note that equations can be numbered using 'tags'. `Markdown` cells can also include images in a variety of ways, one commonly used option is using the syntax `![](path/to/file)`. However, this alternative is not very flexible for controlling the output image (for example the output size of the image). A better alternative is using the *MyST* markdown syntax that allows to control many parameters of the image and its context (size, caption, etc.).

The following unrendered text in a cell is an example of formatted text that includes lists, equations, code, and an image:

````
- Points from a list
    - Different levels of lists
        - text in **bold** and *italics* (or with emphasis)
        - Inline equation: _"the circumference of a circle is_
↪ $2\pi r$"

Equation in its own line:

$$y= mx + b$$


Fragment of code:
``` python
for i in range(0,100):
    print(i)
```
Image:
```{figure} data/Fig_1_1.png
:height: 150px
:name: figure1
:align: center
Magnesium $K_\alpha$ X-ray intensity map from a portion of an␣
↪eclogite polished thin section; image captured in an␣
↪Electron Probe Micro Analyzer.
```
````

This will render as:

- Points from a list
  - Different levels of lists
    * Text in **bold** and *italics* (or with emphasis)
    * Inline equation: "*the circumference of a circle is* $2\pi r$"

Equation in its own line:

$$y = mx + b \tag{1.1}$$

Fragment of code:

```
for i in range(0,100):
    print(i)
```

Image:

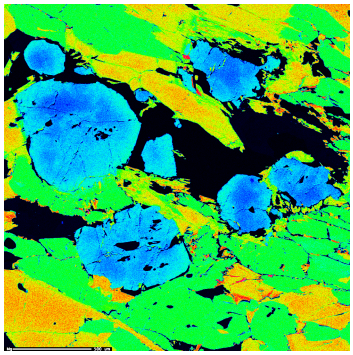

Figure 1.1: Magnesium K_α X-ray intensity map from a portion of an eclogite polished thin section; image captured in an Electron Probe Micro Analyzer.

1.3. Algorithms

Written code in programs is based on algorithms. An algorithm is a list of the sequence of steps (instructions) that must be followed to complete a specific task. Programming consists of encoding these instructions in a language that the computer can interpret and execute.

Worked example 1.1

Write a program to guess a number. For that, use two fundamental programming concepts: (i) the use of variables and constants, and (ii) the use of execution control instructions.
The code needs to generate a random number and ask the user to guess the generated number. The user has a fixed number of attempts. After each attempt, the program needs to inform the user if the guessed number is equal to (in which case the program terminates), below, or above the randomly generated number.

We start by building an algorithm containing a list (sequence) of instructions according to the desired behavior:

```
1. Load the necessary modules.
2. Initialize variables and constants.
3. Start a loop and verify that attempts have not been
   exhausted.
4. Ask the user for a guess.
5. Compare the user's guess with the generated number.
6. If the guess is correct, congratulate the user.
7. If the guess is higher or lower, inform the user of the
   condition of the guess.
```

Note that the instructions that seem necessary when we start the project are explicitly listed above. This sequence can change as we build the program. In some cases, we may need to add instructions; in others, it might be necessary to remove them, or even change the order in the sequence. These changes are almost always necessary in large projects; however, in our small project, we will see that no changes are required.

```
import random #1. functions to generate random numbers
#2.
generated_number = random.randint(1, 20)
attempts = 0
number_found = False
while attempts < 6 and not(number_found): #3.
    number = input("Try to guess the integer: ") #4.
    #5. and 7.
    if int(number) > generated_number:
        message = "My number is greater than " + number
    elif int(number) < generated_number:
        message = "My number is smaller than " + number
    else: #6.
        message = "Congratulations, you guessed the number"
        number_found = True
    print(message)
    attempts += 1
```

1.4. Some Fundamental Programming Concepts in *Python*

1.4.1. Variables and Constants

A variable is an abstraction representing a value that is stored in the computer's memory. To use this value, we assign a name to the variable. The stored value in variables can change during the execution of a program. For example, in the program we built, `attempts` and `number` are examples of variables. A constant is also an abstraction; however, unlike variables, constants do not change their value during program execution. In our example, `generated_number` is a constant. In *Python*, there is no difference between the declaration of variables and constants, i.e., all declared values are variables.

1.4.2. Control Instructions - Loops

Loops are sequences of instructions that are executed an *n* number of times. The number of times the sequence is repeated can be defined initially as part of the loop or can be controlled during program execution. The three most commonly used loops in programming are the **while** loop, the **for** loop, and the **do-while** loop. In the worked example above, there is a **while** loop that controls the number of times the sequence of instructions is repeated inside by changing variable values during execution of the program. However, note that the program places an upper limit on that number (the loop can be executed up to six times).

1.4.3. Control Instructions - Conditional or Alternative

In conditional control instructions (also called Boolean), the execution sequence of a program is altered based on the result of a condition evaluation. If the result of the evaluation is `True` a group of instructions is executed; if the result is `False` another set of instructions is executed. In the example, the number that we introduce is compared with a randomly generated number. If they are the same, the program changes the value of `number_found` to `True` to end the execution. If they are different, the program allows new attempts until the maximum number of attempts is reached.

1.4.4. Functions

A function is a unit of a program that contains a sequence of instructions and operates independently of the rest of the program. The main components of a function block are (Figure 1.2):

- The values that the function receives as input, known as the *parameters*.
- The sequence of instructions coded in a specific programming language.
- The resulting values and/or actions triggered by the instructions. If the function is expected to produce a final resulting value or values, this is called the *return* of the function. The return value is optional, i.e., some functions do not produce one.

In essence, a function is a mini-program; its three components are analogous to the input, process, and output of a program (Figure 1.2). As functions group instructions that can be used to process a set of data in a standard way, they are useful for generalizing code. They also serve to compartmentalize code by taking some sequences of instructions that are used repetitively to locate them outside the main body of the program. Generalization and compartmentalization are central for code reusability, allowing developers to write code that can be used in other programs.

Functions in *Python* are created using the **def** statement (Figure 1.2). For example, the following code snippet is a function created to calculate the factorial of a number:

```
def factorial(n):
    f = 1
    for i in range(1, n + 1):
        f *= i
    return f
```

Variables created inside a function (including the parameters and the result) are called local variables. These variables are only visible inside the function, not from the rest of the program. Functions can take multiple input parameters and produce multiple output values. For example, the following function takes two parameters as arguments and returns two numeric values:

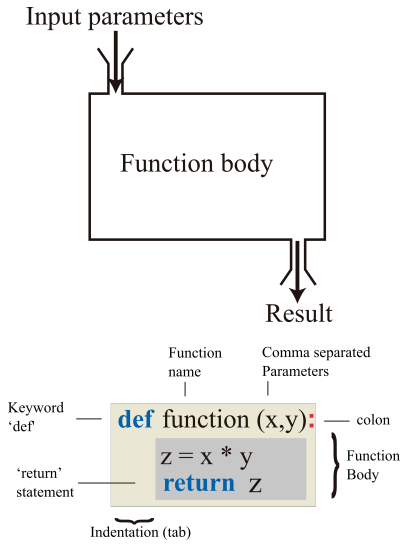

Figure 1.2: Representation (top) and schematic *Python* definition (bottom) of a function

```
def factorials(n1,n2):
    f1 = 1
    f2 = 1
    for i in range(1, n1 + 1):
        f1 *=i
    for i in range(1, n2 + 1):
        f2 *= i
    return f1,f2
```

A function can perform actions without necessarily returning a value, but instead display a printed result, as shown in the following function:

```
def print_factorial(n1,n2):
    f1 = 1
    for i in range(1, n1 + 1):
        f1 *=i
    print(f1, "is the factorial of ", n1)
```

Of course, functions can call other functions. For example, the following function calculates the number of possible combinations between two numbers ($C(m,n) = \frac{m!}{(m-n)!\,n!}$). To do this, it calls the **factorial()** function defined earlier:

```
def comb(m, n):
    fact_m = factorial(m)
    fact_n = factorial(n)
    fact_m_n = factorial(m - n)
    c = fact_m / (fact_n * fact_m_n)
    return m
```

1.5. Data Types in *Python*

Pieces of data used within a program can take a variety of formats. To ensure that the program operates correctly—i.e., that it interprets the value stored in a variable as intended—the programmer has to assign the correct attributes to the data. This is primarily achieved using the predefined data types within the programming language. However, on some special occasions, the programmer needs to define custom data types. Table 1.1 shows some of the most commonly used data types in *Python*.

Table 1.1: Data types in *Python*

Type	Class	Notes	Example
str	String	Immutable	String
unicode	String	Unicode version of str	u'String'
list	Sequence	Mutable; can contain objects of various types	[4, 'String', True]
tuple	Sequence	Immutable, can contain objects of various types	(4, 'String', True)
set	Set	Mutable; no order; contains no duplicates	set(['String', True])
frozenset	Set	Immutable; unordered; contains no duplicates	frozenset([4.0, 'String', True])
dict	Mapping	Group of key-value pairs	{'key1': 1.0, 'key2': False}
int	Integer number	Fixed precision; converted to long in case of overflow	54
long	Integer number	Arbitrary precision	42L
complex	Complex number	Real part and imaginary part j	(4.5 + 3j)
float	Decimal number	Double-precision floating point	3.1415927
bool	Boolean	Boolean true or false value	True or False

1.6. Linear Algebra and *Python* for Scientific Purposes I

1.6.1. *NumPy*

NumPy is a module in *Python* that contains functions for creating and operating on N-dimensional arrays (vectors), which are newly defined data types within the *Python* ecosystem. Functions within *NumPy* are fast (close to that of compiled languages) because they are based on precompiled C code and vectorized. The meaning of this last term becomes clear when examining the following operation between two arrays using a **for** loop:

```
import numpy as np
n = 100
a = np.arange(n)
b = np.arange(n)
sum = 0
for i in range(100):
    sum+= a[i]*b[i]
```

The loop above can be replaced by a *NumPy* vectorized operation:

```
sum = np.sum(a*b)
```

The last piece of code performs an element-by-element operation (broadcasting) on the two arrays, eliminating the need of using a **for** loop.

The list of available functions in *NumPy* is extensive (see the official documentation at https://numpy.org/doc/stable/user/index.html). These functions are subdivided into unary functions (accepts one operand) and binary functions (accepts two operands). In this and the following chapters, we will explore several examples of available functions, including examples of linear algebra operations and random number generation.

1.6.2. Vectors

Vectors describe spatial lines and planes, allowing calculations that explore relationships in multidimensional space. These entities are studied in *linear algebra* under the field of vector spaces. Vectors are mathematical constructions with numeric elements that define n-dimensional coordinates of a point in space. If we think of a vector as an arrow joining the origin of an

n-dimensional space to the point defined by its coordinates, we can determine its *magnitude* and *direction* using its numeric elements. The magnitude represents the distance from the origin, and the direction indicates where the arrowhead is located. In general, an n-dimensional *vector* is a sequence of n numeric elements:

$$\mathbf{v} = [v_1, \ldots, v_n] \tag{1.2}$$

In *Python*, these sequences are known as *arrays*, and there are several ways of constructing them, including the use of the **array()** function from the *NumPy* library. The sequences are generally written horizontally (however, in many cases, their representation must be vertical for use in linear algebra operations). The following code, for example, creates two vectors defined in $\mathbb{R}^3$.

```
import numpy as np
p = np.array([1,2,3])
q = np.array([4,5,6])
```

1.6.3. Inner (Dot) Product

This common operation for vectors takes two vectors of equal dimensions and returns a single value. For the three-dimensional vectors $\mathbf{p}$ and $\mathbf{q}$, their dot product is defined as:

$$\mathbf{p} \cdot \mathbf{q} = (p_1 \cdot q_1) + (p_2 \cdot q_2) + (p_3 \cdot q_3) \tag{1.3}$$

To check that this is the case, we can run the following code:

```
print(np.dot(p,q) == (p[0]*q[0] + p[1]*q[1] + p[2]*q[2]))
```

The above code highlights an important point about *array* indexing in *Python*, the indexing starts at zero instead of one as in common mathematical convention. In the equation above, the first element of vector $\mathbf{p}$ is p_1, but in *Python*, it is referenced as **p[0]**.

1.6.4. Span of a Set of Vectors and Linear Independence

The set of all linear combinations of a given set of vectors is called the span of the vectors. For example, the vectors $\mathbf{s}$ and $\mathbf{t}$, which are linear combinations of $\mathbf{p}$ and $\mathbf{q}$, belong to the span of ($\mathbf{p}$, $\mathbf{q}$):

```
s = 2*p + q
t = 3*p + 5*q
```

A new vector **r** is said to be linearly independent of (**p**, **q**) if **r** does not belong to the span of (**p**, **q**), i.e., **r** cannot be represented by a linear combination of **p** and **q**.

```
r = np.array([6,8,12])
```

Later, we will see that this property is important when analyzing the compositional space. We will also explore how to determine linear independence.

1.6.5. Matrices

In general terms, a matrix is a set of numbers arranged in a rectangular array with rows and columns, like this:

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \tag{1.4}$$

Note that matrices are usually named with uppercase letters. We refer to the individual *elements* of the array using the lowercase equivalent with subscripts indicating the row and column positions. Row indices go from top to bottom, while column indices go from left to right. In standard mathematical notation, indices for rows and columns start at 1. Some programming languages have the same convention.

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{bmatrix} \tag{1.5}$$

In *Python*, you can define a matrix as a *NumPy* two-dimensional **array**, like this:

```
A = np.array([[1,2,3],
              [4,5,6]])
```

Alternatively, you can also use the *NumPy* type **matrix**, which is a specialized subclass of **array**:

```
M = np.matrix([[1,2,3],
               [4,5,6]])
```

However, the **matrix** type is not recommended by the latest documentation of *NumPy*. Instead, using *NumPy* arrays is preferred. All the matrices above have 2 rows and 3 columns, the size or dimensions are an important attribute of a matrix, the **.shape** function extract this information from a defined matrix variable:

```
print(M.shape)  # (2, 3)
```

Matrix elements are accessed using the row and column number (index) of their location. This uses the indicial notation `[i,j]`, where the index *i* represents the row number and index *j* represents the column number. Remember that in *Python*, indices start at 0.

```
print("A(0,0): ", A[0,0])  # A(0,0):  1
```

1.6.6. Block Matrices and Submatrices

Some algorithms require the representation of a large matrix using blocks or *submatrices*. There are also common cases where a *matrix* needs to be constructed using blocks resulting from specific functions in the code. In either case, *Python* offers a variety of functions to facilitate these essential types of calculations.

In the following example, two block matrices are constructed from a *matrix* **A**. The construction of the *submatrices* uses the **range** operators in the two-dimensional case **(:, :)**. The first range operator (before the comma) indicates the rows to be copied, and the second range operator (after the comma) indicates the columns to be copied. The range operators have the syntax **a:b**, where `a` indicates the lower end and `b` the upper end of the range (not inclusive), i.e., **[0:2, 4:6]** indicates rows 0 and 1 and columns 4 and 5.

```
A = np.array([[2, 7, 6, 2, 4, 3],
              [9, 5, 1, 9, 2, 0],
              [4, 3, 8, 1, 0, 9],
              [0, 5, 4, 3, 8, 3]])
A_ul = A[0:2, 0:4]                  # [[2 7 6 2]
                                    #  [9 5 1 9]]
A_ur = A[0:2, 4:6]                  # [[4 3]
                                    #  [2 0]]
A_lr = A[2:4, 4:6]                  # [[0 9]
                                    #  [8 3]]
```

The construction of a *matrix* using *submatrices* is done by applying several functions available in the *NumPy* library. Consider the following code snippet that uses functions to stack rows/columns (or blocks) horizontally and vertically:

```
# Assemble an nd-array from nested lists of blocks.
B = np.block([A_ur, A_ul])
# Vertical stack
C = np.vstack([A_lr, A_ur])
# Stack 1-D arrays as columns
D = np.c_[A_l_stack, np.zeros((9))]
# Stack 1-D arrays as rows
E = np.r_[A_l_stack, np.zeros((1,3))]
```

1.6.7. Matrix Transpose

The *transpose* operation of a matrix changes the orientation of its rows and columns. This operation is indicated using a superscript T:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix} \tag{1.6}$$

In *Python*, both the *NumPy* **array** and the *NumPy* **matrix** have a function **T**:

```
A = np.array([[1,2,3],
              [4,5,6]])
A_transpose = A.T  # [[1, 4],
                   #  [2, 5],
                   #  [3, 6]]
```

1.6.8. Matrix-Matrix Multiplication

Matrix multiplication involves the calculation of the *dot product* of rows and columns. The algorithm for performing this operation follows the dot product of rows and columns (*RC* rule - multiply ***R***ows of the first matrix with ***C***olumns of the second matrix), in mathematical terms:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} \tag{1.7}$$

$$c_{ij} = \sum_{k=1}^{p} a_{ik} b_{kj} \tag{1.8}$$

For the matrices to be *conforming* to the dot product operation, the number of ***columns*** in the first matrix must match the number of ***rows*** in the second matrix. The following *Python* code snippet performs the dot product between two matrices. The result is verified by comparing the first element of the resulting matrix **C** and the dot product of the first row of **A** with the first column of **B**:

```
A = np.array([[1,2,3],
              [4,5,6]])
B = np.array([[9,8],
              [7,6],
              [5,4]])
C = np.dot(A,B)       #[[ 38  32]
                      # [101  86]]
# check the first individual operation
print(np.dot(A[0],B[:,0]) == C[0,0]) # True
```

1.6.9. Inverse of a Matrix

The inverse of a square $n \times n$ matrix is another $n \times n$ matrix. The inverse of **B** is written as $\mathbf{B^{-1}}$. Note that the dot product of a matrix with its inverse is always the identity matrix. The inverse of a 2×2 matrix is given by the following formula:

$$\mathbf{B^{-1}} = \begin{bmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{bmatrix}^{-1} = \frac{1}{b_{1,1} \times b_{2,2} - b_{1,2} \times b_{2,1}} \begin{bmatrix} b_{2,2} & -b_{1,2} \\ -b_{2,1} & b_{1,1} \end{bmatrix} \quad (1.9)$$

The following code snippet calculates the inverse of a 2×2 matrix:

```
B = np.array([[6,2],
              [1,2]])
B_inv = 1/(6*2-2*1)*np.array([[2,-2],
                              [-1,6]])
print(np.linalg.inv(B)) #[[ 0.2 -0.2]
                        # [-0.1  0.6]]
```

1.6.10. The Determinant of a Matrix

The *determinant* is a property (scalar quantity) of a square matrix that determines if the matrix is invertible. This property is denoted as $det(\mathbf{B})$ or $|\mathbf{B}|$. From the previous statement, it follows that the existence of $\mathbf{B^{-1}}$ is directly

related to the determinant of **B**; if $det(\mathbf{B}) = 0$, the matrix is non-invertible and it is said to be *singular*.

The determinant of an 2×2 square matrix is the product of the diagonal elements minus the product of the off-diagonal elements:

$$|\mathbf{B}| = \begin{vmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{vmatrix} = b_{1,1} \times b_{2,2} - b_{1,2} \times b_{2,1} \tag{1.10}$$

For example:

$$\begin{vmatrix} 6 & 2 \\ 1 & 2 \end{vmatrix} = 6 \times 2 - 1 \times 2 = 10 \tag{1.11}$$

For larger square matrices, the calculation of the determinant is based on the definition of minors and cofactors. A minor of a matrix is the determinant of a submatrix obtained by removing a specific row and a specific column from the original matrix. Minors are symbolized using subindices to indicate which are the column and the row missing in the submatrix. Consider the following matrix **A**:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \tag{1.12}$$

For this matrix, the minor $M_{1,2}$ is the determinant of the submatrix obtained by removing row 1 and column 2:

$$M_{1,2} = \begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} = a_{2,1} \times a_{3,3} - a_{2,3} \times a_{3,1} \tag{1.13}$$

Each element in a $n \times n$ matrix ($n > 2$) has a corresponding minor. A matrix of minors can be formed by calculating the minors for each element. The cofactors are defined as the minors multiplied by -1^{i+j}. For example, for row 1 and column 1 the cofactor is the minor ($-1^{1+1} = 1$) and for row 1 and column 2 the cofactor is minus the minor ($-1^{1+2} = -1$). These definitions allow us to derive a general algorithm to calculate the determinant of any square matrix:

1. Select any row or column in the matrix. It does not matter which row or column is used—the result will be the same; however, for some rows or columns the calculation will be faster (i.e., those that have more zero elements).

2. The determinant is the sum of the terms obtained by multiplying every element in the selected row or column by its cofactor.

3. The determinant of a $n \times n$ matrix ($n > 2$) can be calculated inductively, that is, the determinant of a 3×3 matrix is calculated using 2×2 determinants, the determinant of a 4×4 matrix is calculated using 3×3 determinants, and so on.

For a 3×3 matrix, selecting row 1, the determinant will be the result of the following operation:

$$|\mathbf{A}| = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = a_{1,1} \begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} - a_{1,2} \begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} + a_{1,3} \begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix} \tag{1.14}$$

One way to reduce computations when calculating the determinant is by using the LU decomposition (see Section 1.6.16), because $det(\mathbf{A}) = \pm det(\mathbf{U})$ and the determinant of an echelon form matrix is the product of its main diagonal elements (the ± sign is negative if an odd number of row interchanges occurred).

The inverse of a matrix can be calculated using the cofactor method:

$$A^{-1} = \frac{1}{det(\mathbf{A})} adj(\mathbf{A}) \tag{1.15}$$

where $adj(\mathbf{A})$ is the adjoint of matrix $\mathbf{A}$ defined as the transpose of the matrix of cofactors of $\mathbf{A}$.

Worked example 1.2

Determine if the following matrix $\mathbf{C}$ is invertible by calculating its determinant. If the matrix is invertible, calculate $\mathbf{C}^{-1}$ using the cofactor method.

$$\mathbf{C} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 1 & 3 \\ 3 & 2 & 4 \end{bmatrix}$$

1. Calculate the determinant of $\mathbf{C}$. By selecting row 1:

$$|\mathbf{C}| = 1 \begin{vmatrix} 1 & 3 \\ 2 & 4 \end{vmatrix} - 2 \begin{vmatrix} 2 & 3 \\ 3 & 4 \end{vmatrix} + 1 \begin{vmatrix} 2 & 1 \\ 3 & 2 \end{vmatrix}$$

$$|\mathbf{C}| = 1(1 \times 4 - 3 \times 2) - 2(2 \times 4 - 3 \times 3) + 1(2 \times 2 - 1 \times 3) = 1$$

$|\mathbf{C}| \neq 0$, therefore the matrix is invertible.

Worked example 1.2 (cont.)

2. Compute the matrix of cofactors. The cofactors of row 1 can be visualized from the calculation of the determinant in numeral 1 ([−2 1 1]). As another example of the procedure, the cofactors for elements in row 2 are shown next:

   - For $c_{2,1}$ the cofactor is:

$$(-1)^{2+1}\begin{vmatrix} 2 & 1 \\ 2 & 4 \end{vmatrix} = -6$$

   - For $c_{2,2}$ the cofactor is:

$$(-1)^{2+2}\begin{vmatrix} 1 & 1 \\ 3 & 4 \end{vmatrix} = 1$$

   - For $c_{2,3}$ the cofactor is:

$$(-1)^{2+3}\begin{vmatrix} 1 & 2 \\ 3 & 2 \end{vmatrix} = 4$$

The final matrix of cofactors is:

$$\begin{bmatrix} -2 & 1 & 1 \\ -6 & 1 & 4 \\ 5 & -1 & -3 \end{bmatrix}$$

3. Calculation of $\mathbf{C^{-1}}$:

$$\mathbf{C^{-1}} = \frac{1}{1}\begin{bmatrix} -2 & 1 & 1 \\ -6 & 1 & 4 \\ 5 & -1 & -3 \end{bmatrix}^T = \begin{bmatrix} -2 & -6 & 5 \\ 1 & 1 & -1 \\ 1 & 4 & -3 \end{bmatrix}$$

4. Verify results with *NumPy*:

```
import numpy as np
C = np.array([[1, 2, 1],
              [2, 1, 3],
              [3, 2, 4]])
np.linalg.det(C), np.linalg.inv(C)
```

1.6.11. The Rank of a Matrix

The column space of an $m \times n$ matrix $\mathbf{A}$ is the vector space spanned by the columns of $\mathbf{A}$ (remember that the span provides information about linear independence). Similarly, the row space of an $m \times n$ matrix $\mathbf{A}$ is the vector space spanned by the columns of $\mathbf{A}^{\mathrm{T}}$. The dimension of the column space of $\mathbf{A}$ represents the rank of the matrix. This value is always less than or equal

to the minimum of n and m. The dimension of the row space of $\mathbf{A}$ is also equal to the rank of matrix $\mathbf{A}$. The rank of a matrix represents the number of independent rows or columns in the matrix. In the following example, matrix $\mathbf{A}$ has a rank equals to two, the matrix has two independent rows and two independent columns. It is easily seen that the third row is obtained by summing the first and the second row.

```
A = np.array([[1, 2, 4, 1, -3],
              [2, 4, 0, 0, -6],
              [3, 6, 4, 1, -9]])
rankA = np.linalg.matrix_rank(A)  # 2
```

Note that the above matrix is said to be rank deficient since $rank(\mathbf{A}) < min(m, n)$.

1.6.12. The Null Space of a Matrix

In an $m \times n$ matrix $\mathbf{A}$, the set of vectors in $\mathbb{R}^n$ for which $\mathbf{A} \cdot x = 0$ is called the null space of $\mathbf{A}$. It is denoted by $N(\mathbf{A})$. The operation of finding the null space is equivalent to solving the system $\mathbf{A} \cdot x = 0$. In the following example the null space of a matrix is obtained by using the **linalg.null_space** function from *SciPy*, then the result is verified using the dot product of the first row with the null space vector.

```
import scipy.linalg as la
A = np.array([[1, 3, 2],
              [2, -3, -5]])
nullA = la.null_space(A) #[[ 0.57735027]
                         # [-0.57735027]
                         # [ 0.57735027]]
# Convert the resulting matrix in a vector
nullA_vec = nullA[:,0]     #[ 0.57735027 -0.57735027  0.57735027]
# test the result with the first row
print(np.round(np.dot(A[0],nullA_vec), 15) == 0.0)   # True
```

1.6.13. Summary of Tests for Singularity

There are several conditions that indicate whether an $n \times n$ $\mathbf{A}$ matrix is invertible or nonsingular; these conditions represent different tests for invertibility. A matrix is nonsingular if any of the following conditions hold:

- $det(\mathbf{A}) \neq 0$.

- $rank(\mathbf{A}) = n$, $\mathbf{A}$ is a full rank matrix.
- $\mathbf{A} \cdot x = 0$ implies that $x = 0$ is the unique solution. The null space of A is empty.
- The rows (or columns) of $\mathbf{A}$ are linearly independent.

1.6.14. Solution of Linear Systems

A full-rank square matrix (where all rows are independent) representing a system of linear equations ($\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$) can be used to solve the linear system; the unique solution of the system is given by:

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \tag{1.16}$$

As an example, let us consider the following linear system:

$$2x_1 + 4x_2 = 18 \tag{1.17}$$

$$6x_1 + 2x_2 = 34 \tag{1.18}$$

In matrix form, the dot product of the coefficient matrix $\mathbf{A}$ with the solution vector $\mathbf{x}$ equals the vector $\mathbf{b}$:

$$\begin{bmatrix} 2 & 4 \\ 6 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 18 \\ 34 \end{bmatrix} \tag{1.19}$$

Since $\mathbf{A}$ is nonsingular (see the test below), the solution of the linear system is given by equation (1.16). The following code tests the matrix for singularity and finds the solution to the linear system:

```
A = np.array([[2,4],
              [6,2]])
print(la.det(A) != 0) # True
b = np.array([18, 34])
x = np.dot(la.inv(A), b)   #[5. 2.]
```

However, the above technique is not the most efficient for solving linear systems; the set of techniques used to solve a system of linear equations (or inequalities) and to maximize or minimize linear functions is known as *linear programming*. There are three necessary steps to use these techniques: (i) Defining variables, (ii) Identifying constraints, and (iii) Determining an objective function. *NumPy* offers many *linear programming* algorithms; for example, it provides the **linalg.solve** function for linear systems that can be

represented by a full-rank matrix (where all rows are linearly independent). In other cases, the **linalg.lstsq** function calculates the best solution of the system using the least squares method.

In the following example, a system of three linear equations is solved for one right-hand side vector; the solution is tested against the right-hand side vector:

```
import scipy.linalg as la
A = np.array([[ 4,  3, 2],
              [-2,  2, 3],
              [ 3, -5, 2]])
b = np.array([25, -10, -4])
x = la.solve(A,b)              # [ 5.  3. -2.]
np.dot(A,x) == b               # True
```

1.6.15. Transformations

Many scientific computations involve transforming vectors to a new reference frame. Some of the commonly needed operations on data include rotation, scaling, and translation. These transformations of vectors are performed by multiplying a transformation matrix with the vectors. The transformation matrix is the operand (function) that maps the coordinates of the vectors to the new reference frame.

1.6.15.1. Linear Transformations

A *linear transformation* is an operation on vectors that respects the underlying linear structure of the vectors. We can define a transformation T as the operation:

$$T(\mathbf{v}) = \mathbf{A} \cdot \mathbf{v} \tag{1.20}$$

For the transformation to be linear, for any two vectors ($\mathbf{x}$ and $\mathbf{y}$) of the vector space, the following holds (a and b are scalars):

$$T(a\mathbf{x} + b\mathbf{y}) = aT(\mathbf{x}) + bT(\mathbf{y}) = a\mathbf{A} \cdot \mathbf{x} + b\mathbf{A} \cdot \mathbf{y} \tag{1.21}$$

For example, consider the matrix $\mathbf{A}$ and the vector $\mathbf{v}$:

$$\mathbf{A} = \begin{bmatrix} 1.5 & 0.0 \\ 3.1 & 2.4 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 2.3 \\ 0.5 \end{bmatrix} \tag{1.22}$$

This is:

$$\begin{bmatrix} 1.5 & 0.0 \\ 3.1 & 2.4 \end{bmatrix} \cdot [2.3, 0.5] = \begin{bmatrix} 3.45 \\ 8.33 \end{bmatrix} \tag{1.23}$$

in *Python*:

```
v = np.array([2.3,0.5])
A = np.array([[1.5,0],
              [3.1,2.4]])
T = np.dot(A,v)       # [3.45, 8.33]
```

In this case, the transformation T takes a 2-dimensional vector and produces a new 2-dimensional vector:

$$T : \mathbb{R}^2 \rightarrow \mathbb{R}^2 \tag{1.24}$$

In a more general case, the resulting vector may have a different number of dimensions than the original vector (the matrix transforms the vector from one space to another):

$$T : \mathbb{R}^n \rightarrow \mathbb{R}^m \tag{1.25}$$

The following *Python* examples show transformations that reduce and increase the dimension of a vector:

```
v = np.array([2.3,0.5])
A = np.array([[2,3],
              [5,2],
              [1,1]])
# increasing the space of the vector
transformed_v = np.dot(A,v) # [ 6.1 12.5  2.8]
v = np.array([2.3,0.5, 0.8])
A = np.array([[1.5,0,1.0],
              [3.1,2.4,4.5]])
# reducing the space of the vector
transformed_v = np.dot(A,v) # [ 4.25 11.93]
```

Note that linear transformations can change both the magnitude and direction of a vector. Two simple cases can exemplify these effects. In the first example below, the transformation scales the vector, changing only its magnitude; in the second example, the transformation rotates the vector, changing only its direction:

- 1

$$\mathbf{A} = \begin{bmatrix} 3.0 & 0.0 \\ 0.0 & 3.0 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 2.5 \\ 0.0 \end{bmatrix} \tag{1.26}$$

$$\mathbf{A} \cdot \mathbf{v} = \begin{bmatrix} 7.5 \\ 0.0 \end{bmatrix} \tag{1.27}$$

- 2

$$\mathbf{A} = \begin{bmatrix} 0.0 & -1.0 \\ -1.0 & 0.0 \end{bmatrix} \quad \mathbf{v} = \begin{bmatrix} 2.5 \\ 0.0 \end{bmatrix} \tag{1.28}$$

$$\mathbf{A} \cdot \mathbf{v} = \begin{bmatrix} 0.0 \\ -2.5 \end{bmatrix} \tag{1.29}$$

As we will see later, linear transformations are central to the analysis of the compositional space.

1.6.15.2. Affine Transformations

A transformation is affine if it preserves lines and parallelism; all linear transformations are affine, but not all affine transformations are linear. An affine transformation consists of multiplying a vector by a matrix and adding a compensation vector, sometimes called a *bias*; like this:

$$T(\mathbf{v}) = \mathbf{A} \cdot \mathbf{v} + \mathbf{b} \tag{1.30}$$

For example:

$$\begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -2 \\ -6 \end{bmatrix} = \begin{bmatrix} 5 \\ -2 \end{bmatrix} \tag{1.31}$$

This type of transformation is actually the basis for linear regression. The matrix defines the *features*, the first vector contains the *coefficients* and the bias vector acts as the *intercept*.

```
v = np.array([2.3,0.5, 0.8])
A = np.array([[1.5,0,1.0],
              [3.1,2.4,4.5],
              [5.1,0.2,-7.1]])
b = np.array([0.3,1.2,-1.5])
t = np.dot(A,v) + b # [ 4.55 13.13  4.65]
```

1.6.16. Decompositions

Matrix decompositions include a number of operations applied to a matrix to express it as a product of matrices—i.e., factorization of a matrix. Matrix decompositions are widely used in several matrix algorithm implementations, for example, solving systems of linear equations. Some commonly used decompositions include *LU*, *QR*, and *SVD*. *NumPy* and *SciPy* provide implementations of most commonly used matrix decompositions. We will explore some details of the *LU* and *SVD* in more detail in the following chapters, as they are essential for upcoming topics.

1.6.16.1. *LU* Decomposition

LU decomposition is the process of decomposing a matrix $\mathbf{A}$ into a product $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$, where $\mathbf{L}$ is a unit lower triangular matrix (all of its main diagonal entries are ones) and $\mathbf{U}$ is an upper triangular matrix. Matrix $\mathbf{U}$ results from Gaussian elimination—i.e, the result of applying row operations to make the elements below the diagonal to zero. Matrix $\mathbf{L}$ is a representation of the factors used during the elimination steps, and thus, it can be used to recover the original matrix $\mathbf{A}$.

Worked example 1.3

Perform Gaussian elimination on the matrix $\mathbf{A}$ and save the applied operations during the elimination of elements below the main diagonal to get the matrix $\mathbf{L}$.

$$\mathbf{A} = \begin{bmatrix} 5.0 & 1.0 & 3.0 \\ 10.0 & -2.0 & 8.0 \\ -5.0 & -5.0 & 3.0 \end{bmatrix}$$

The following are the needed operations to clear elements below the diagonal:

- $Row\ 2 = Row\ 2 - 2 * Row\ 1$
- $Row\ 3 = Row\ 3 + Row\ 1$
- $Row\ 3 = Row\ 3 - Row\ 2$

In matricial form, these operations are:

$$\mathbf{Op_1} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -2.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\mathbf{Op_2} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 1.0 & 0.0 & 1.0 \end{bmatrix}$$

Worked example 1.3 (cont.)

$$\mathbf{Op_3} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & -1.0 & 1.0 \end{bmatrix}$$

And gathering all operations in a single matrix:

$$\mathbf{L_p} = Op_3 \cdot (Op_2 \cdot Op_1) = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -2.0 & 1.0 & 0.0 \\ 3.0 & -1.0 & 1.0 \end{bmatrix}$$

The upper triangular matrix is computed as:

$$\mathbf{U} = \mathbf{L_p} \cdot \mathbf{A} = \begin{bmatrix} 1.0 & 0.0 & 0.0 \\ -2.0 & 1.0 & 0.0 \\ 3.0 & -1.0 & 1.0 \end{bmatrix} \cdot \begin{bmatrix} 5.0 & 1.0 & 3.0 \\ 10.0 & -2.0 & 8.0 \\ -5.0 & -5.0 & 3.0 \end{bmatrix} = \begin{bmatrix} 5.0 & 1.0 & 3.0 \\ 0.0 & -4.0 & 2.0 \\ 0.0 & 0.0 & 4.0 \end{bmatrix}$$

The above relation can be expressed as:

$$\mathbf{L_p^{-1}} \cdot \mathbf{U} = \mathbf{A}$$

Having $\mathbf{L} = \mathbf{L_p^{-1}}$

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A}$$

In *Python*:

```
import numpy as np
A = np.array([[5,   1, 3],
              [10, -2, 8],
              [-5, -5, 3]])
row2 = A[1,:] - 2*A[0,:]       # [0 -4  2]
row3 = A[2,:] - (-1)*A[0,:]    # [0 -4  6]
row3 = row3 - row2             # [0  0  4]
op_1 = np.array([[ 1, 0, 0],
                 [-2, 1, 0],
                 [ 0, 0, 1]])
op_2 = np.array([[ 1, 0, 0],
                 [ 0, 1, 0],
                 [ 1, 0, 1]])
op_3 = np.array([[ 1, 0, 0],
                 [ 0, 1, 0],
                 [ 0,-1, 1]])
Lp = np.dot(op_3, np.dot(op_2,op_1))
U = np.dot(Lp,A)
L = np.linalg.inv(Lp)
```

Worked example 1.3 (cont.)

By multiplying L and U, we obtain the original matrix A. Note that the matrix can be reordered (row interchange) and in doing so we can get a different but equivalent solution. The row exchange is needed when the element in a diagonal (called the *pivot*) for a particular row is zero.

1.6.16.2. Eigenvalue Decomposition and Singular Value Decomposition (*SVD*)

Suppose we have a linear transformation in the form $\mathbf{A} \cdot x$. If this transformation satisfies the following equation:

$$\mathbf{A} \cdot x = \lambda \ x \tag{1.32}$$

the λ value is known as the eigenvalue of matrix $\mathbf{A}$, and the associated vector x is called the eigenvector corresponding to the λ value. The interpretation of this operation is that, for some vectors, the effect of the transformation is only scaling (stretching, compressing, or flipping), with λ acting as the scaling factor. Eigenvalues and eigenvectors can be derived using the *characteristic equation*:

$$det(\mathbf{A} - \lambda \ \mathbf{I}) = 0 \tag{1.33}$$

Using an appropriate basis, any matrix $\mathbf{A}$ can be expressed as a diagonal matrix of eigenvalues; from this, it follows that a matrix $\mathbf{A}$ can be decomposed into a diagonal matrix using two different bases (the sets of left and right singular vectors), this is known as the *SVD* decomposition:

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T \tag{1.34}$$

where $\mathbf{A}$ is the input $m \times n$ matrix, $\mathbf{U}$ are the left singular vectors, Σ is a diagonal matrix with singular values (square roots of the eigenvalues) for $\mathbf{A}$, and $\mathbf{V}$ contains the right singular vectors.

The *SVD* of $\mathbf{A}$ is related to the eigenvalue decomposition of the matrix $\mathbf{A}^T \cdot \mathbf{A}$:

1. Calculate $\mathbf{A}^T \cdot \mathbf{A}$ and compute the eigenvalue decomposition of $\mathbf{A}^T \cdot \mathbf{A}$.

   a. The eigenvalues are obtained by calculating the determinant of $\mathbf{A}^T \cdot \mathbf{A} - \lambda \ \mathbf{I}$ and solving the resulting characteristic polynomial.

   b. The eigenvectors for each eigenvalue are derived from $\mathbf{A}^T \cdot \mathbf{A} - \lambda \ \mathbf{I}$ formed for each eigenvalue. The resulting matrix is reduced to row

echelon form applying elementary row operations. The null space of the row echelon form, converted into a unit vector, is the eigenvector of the used eigenvalue.

2. Σ is a non-negative diagonal matrix, where each element is the square root of an eigenvalue, sorted in descending order. These are the nonzero singular values of $\mathbf{A}$.

3. The columns of $\mathbf{V}$ are the eigenvectors.

4. $\mathbf{U}$ is calculated from $\sigma_i u_i = \mathbf{A}v_i$ that results in $u_i = 1/\sigma_i \mathbf{A}v_i$.

The matrix $\mathbf{A}^T \cdot \mathbf{A}$ is known as the covariance matrix, which has a well-known interpretation in statistics. However, this algorithm is unstable, and an alternative way to compute *SVD* is to calculate the eigenvalue decomposition of:

$$\mathbf{H} = \begin{bmatrix} 0 & \mathbf{A}^T \\ \mathbf{A} & 0 \end{bmatrix} \tag{1.35}$$

Worked example 1.4

Compute the Singular Value Decomposition (*SVD*) of the following matrix:

$$\mathbf{A} = \begin{bmatrix} 25 & 3 \\ 0 & 16 \end{bmatrix}$$

---

1. Compute the eigenvalue decomposition of $\mathbf{A}^T \cdot \mathbf{A}$.

$$\mathbf{A}^T \cdot \mathbf{A} - \lambda\, \mathbf{I} = \begin{bmatrix} 625 - \lambda & 75 \\ 75 & 265 - \lambda \end{bmatrix}$$

The determinant of an 2×2 square matrix is the product of the diagonal minus the product of the off-diagonal elements:

$$|\mathbf{A}^T \cdot \mathbf{A} - \lambda \mathbf{I}| = (625 - \lambda) \cdot (265 - \lambda) - (75) \cdot (75)$$

$$\lambda^2 - 890\lambda + 160000 = (\lambda - 640) \cdot (\lambda - 250)$$

Thus, the two eigenvalues are 640 and 250.

2. Derive the eigenvectors:

- For $\lambda = 640$

$$\mathbf{A}^T \cdot \mathbf{A} - 640\, \mathbf{I} = \begin{bmatrix} -15 & 75 \\ 75 & -375 \end{bmatrix}$$

Row 2 = Row 2 + 5*Row 1

$$\begin{bmatrix} -15 & 75 \\ 0 & 0 \end{bmatrix}$$

Worked example 1.4 (cont.)

Row 1 = Row 1 / -15

$$\begin{bmatrix} 1 & -5 \\ 0 & 0 \end{bmatrix}$$

A null space for this matrix comes from $x_1 - 5\,x_2 = 0$, which gives:

$$nullspace \begin{bmatrix} 1 & -5 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 5 \\ 1 \end{bmatrix}$$

The eigenvector corresponding to the eigenvalue of 640 is the unit vector:

$$\begin{bmatrix} 5/\sqrt{26} \\ 1/\sqrt{26} \end{bmatrix}$$

- For $\lambda = 250$

$$\mathbf{A}^T \cdot \mathbf{A} - 250\,\mathbf{I} = \begin{bmatrix} 375 & 75 \\ 75 & 15 \end{bmatrix}$$

Row 1 = Row 1 - 5*Row 2

$$\begin{bmatrix} 0 & 0 \\ 75 & 15 \end{bmatrix}$$

Row 2 = Row 2 / 15

$$\begin{bmatrix} 0 & 0 \\ 5 & 1 \end{bmatrix}$$

A null space for this matrix comes from $5\,x_1 + x_2 = 0$, which gives:

$$nullspace \begin{bmatrix} 0 & 0 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 5 \end{bmatrix}$$

The eigenvector corresponding to the eigenvalue of 250 is the unit vector:

$$\begin{bmatrix} -1/\sqrt{26} \\ 5/\sqrt{26} \end{bmatrix}$$

The matrix with the right singular vectors is:

$$\mathbf{V} = \begin{bmatrix} 5/\sqrt{26} & -1/\sqrt{26} \\ 1/\sqrt{26} & 5/\sqrt{26} \end{bmatrix}$$

3. The singular values are $\sqrt{640}$ and $\sqrt{250}$, then:

$$\Sigma = \begin{bmatrix} \sqrt{640} & 0 \\ 0 & \sqrt{250} \end{bmatrix}$$

4. Compute the left singular vectors:

$$u_1 = \frac{1}{\sqrt{640}}\mathbf{A} \cdot \begin{bmatrix} 5/\sqrt{26} \\ 1/\sqrt{26} \end{bmatrix} = \begin{bmatrix} 8/\sqrt{65} \\ 1/\sqrt{65} \end{bmatrix}$$

$$u_2 = \frac{1}{\sqrt{250}}\mathbf{A} \cdot \begin{bmatrix} -1/\sqrt{26} \\ 5/\sqrt{26} \end{bmatrix} = \begin{bmatrix} -1/\sqrt{65} \\ 8/\sqrt{65} \end{bmatrix}$$

Worked example 1.4 (cont.)

then:

$$\mathbf{U} = \begin{bmatrix} 8/\sqrt{65} & -1/\sqrt{65} \\ 1/\sqrt{65} & 8/\sqrt{65} \end{bmatrix}$$

5. Test the result:

```
import numpy as np
V = np.array([[5/(26**0.5), -1/(26**0.5)],
              [1/(26**0.5),  5/(26**0.5)]])
S = np.array([[640**0.5, 0       ],
              [0       , 250**0.5]])
U = np.array([[8/(65**0.5), -1/(65**0.5)],
              [1/(65**0.5),  8/(65**0.5)]])
print(np.dot(U, np.dot(S, V.T)))
```

Note that *NumPy* has functions to perform the computation of the Eigenvalue and the *SVD* decompositions:

```
import numpy as np
A = np.array([[25,3],
              [0,16]])
print(np.linalg.eig(np.dot(A.T, A)))
print(np.linalg.svd(A))
```

1.6.17. Some Useful *NumPy* Functions

1. `numpy.where()`: Used to search within arrays for elements that meet one or more conditions. The function returns two vectors, representing the row and column indices of the elements that satisfy the conditions. In the *Python* example below, we look for data within a matrix fitting a specific criterion (values greater than 10 and smaller than 30). This produces two index arrays with the i, j values of the elements fitting the condition. The code then creates a matrix of zeros and ones, values that fit the criteria are replaced with ones, and values that do not fit the criteria are replaced with zeros:

```
A = np.array([[ 0,  2,  4,  6, 18, 20, 22, 24, 36, 38, 40, 42],
              [ 1,  3,  5,  7, 19, 21, 23, 25, 37, 39, 41, 43],
              [ 2,  4,  6,  8, 20, 22, 24, 26, 38, 40, 42, 44],
              [11, 13, 15, 17, 29, 31, 33, 35, 47, 49, 51, 53],
```

(continues on next page)

(continued from previous page)

```
              [20, 22, 24, 26, 38, 40, 42, 44, 56, 58, 60, 62],
              [19, 21, 23, 25, 37, 39, 41, 43, 55, 57, 59, 61],
              [18, 20, 22, 24, 36, 38, 40, 42, 54, 56, 58, 60],
              [ 9, 11, 13, 15, 27, 29, 31, 33, 45, 47, 49, 51],
              [10, 12, 14, 16, 28, 30, 32, 34, 46, 48, 50, 52]])
A_g10_l30 = np.where(np.logical_and(A>10, A<30))
A_filter = np.where(np.logical_and(A>10, A<30), 1, 0)
```

2. `numpy.ix_()`: Used to build a mesh of indices. For example, to extract data of matrix **A** (as defined above) from rows 1 and 3 and columns 2 and 5 (a total of four elements), we could write in *Python*:

```
ix_rows = [1,3]
ix_cols = [2,5]
net_ix = np.ix_(ix_rows, ix_cols)
# Build a new matrix with elements [1,2], [1,5], [3,2], and [3,5]
B = A[net_ix]
```

1.7. *Python* for Scientific Purposes II

1.7.1. *Matplotlib*

Matplotlib is a *Python* package for 2D plotting that provides functions to create a wide variety of plot types. The package has many plotting capabilities and allows customization of plots, making its use very flexible. Results can then be saved in several output formats (PDF, PNG, PS, etc.), or visualized in interactive sessions as in notebooks. A great feature of *Matplotlib* is its integration with *LaTeX*, allowing users to include equations within the plot. The following example shows how *Matplotlib* is used to plot a curve representing the function $\sin x^2$ (Figure 1.3). First, a range of values in `x` is created using the **numpy.linspace** function, then `y` values are calculated with $y = \sin x^2$, and finally the **matplotlib.pyplot.plot** function is used to plot results. Note that the function takes as the first parameters the `x` and `y` values, then, takes other parameters to format the plot (in this case just two parameters). The label parameter contains a *LaTeX* equation, the use of the **matplotlib.pyplot.legend** function instructs the interpreter to display the legend on the plot.

```
import matplotlib.pyplot as plt
x = np.linspace(0, 2 * np.pi, 2000)
y = np.sin(x**2)
plt.plot(x, y, color="green",
         label=r'Function: $\sin x^2$')
plt.legend()
plt.show()
```

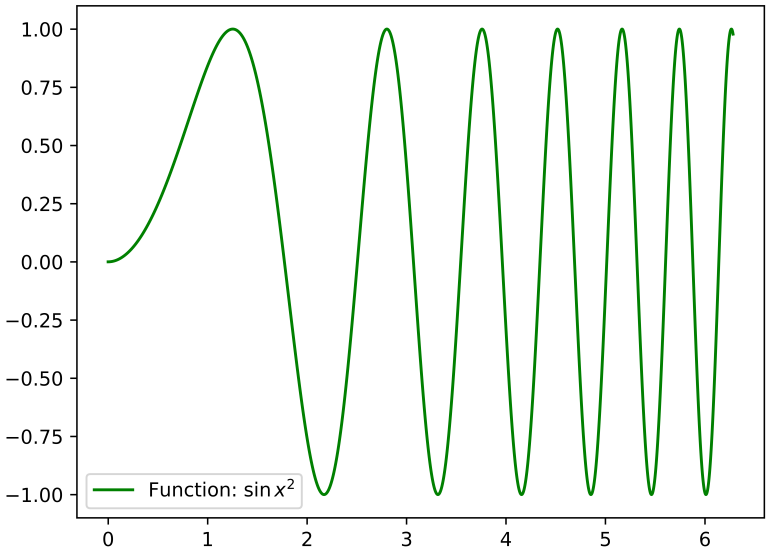

Figure 1.3: A simple graph produced with the *Matplotlib* package.

1.7.2. *SciPy*

SciPy is a library of functions that provides additional functionality beyond what is offered by *NumPy*. For example, it has routines to solve basic differential equations and do numerical integration, for optimization, to perform signal processing and image processing, and for generation of random numbers. For example, the *SciPy* module has a function to perform the *LU* decomposition described above:

```
from scipy.linalg import lu
l,u = lu(A, permute_l = True)
```

1.7.3. Optimization

1.7.3.1. Maximum and Minimum of a Function

The classic example of an optimization routine deals with finding a function's minimum and maximum values. To illustrate the complexity of this operation, we will create a function with more than one minimum. Figure 1.4 shows a function with four minimum values in the range $-10 < x < 10$. To generate this figure, we can use the following *Python* code:

```
import numpy as np
import matplotlib.pyplot as plt
def complex_function(x):
    return x**2 + 20*np.cos(x) - np.sin(x)
x = np.arange(-10, 10, 0.1)
fig, ax = plt.subplots()
ax.plot(x, complex_function(x));
```

To find the minima, we can use the *optimize* module from the *SciPy* library calling the optimization routine with the **complex_function** as an argument. Note that, initially, we do not ask for a specific method and the optimization routine uses the default method (the *Nelder Mead* algorithm). The routine takes as input parameters the function to be optimized and an initial value. Also note that the function has three local minima and one global minimum in the range -10 to 10, the one that the routine finds depends on the provided initial value.

```
from scipy import optimize
print(optimize.minimize(complex_function, x0=0))
# fun: -11.34; x: 2.807
```

The result from the optimization routine includes information about the successful termination, the minimum found (`x` and `y`), and two arrays (*Jacobian* and *Hessian*). The *Jacobian* contains all the first-order partial derivatives of the (multivariate) function, i.e., an array of values describing the slope of the function with respect to each of the variables (in this case only one). The *Hessian* is a square 2-D array (a matrix) with the second-order partial derivatives, it describes the local curvature of the function. If we call the minimization function with an initial value towards one of the extremes in the plot above, we will get a local minimum:

```
print(optimize.minimize(complex_function, x0=-10))
# fun: 61.00; x: -8.467
```

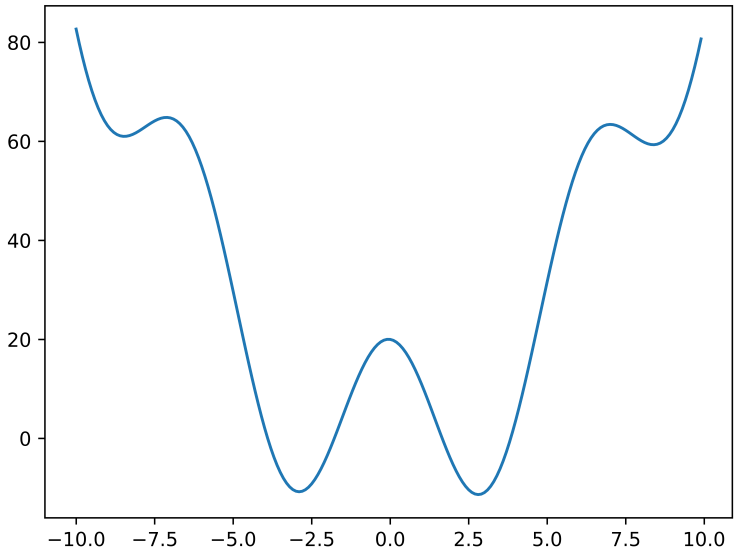

Figure 1.4: y vs. x graph of the function $y = x^2 + 20\ \cos x - \sin x$.

The function can also be called specifying the desired algorithm to use during minimization:

```
print(optimize.minimize(complex_function, x0=0, method="L-BFGS-B"))
```

1.7.3.2. Roots of Nonlinear Systems

The problem is stated as follows: given a continuous nonlinear function $f(x)$, the goal is to find a set of x values satisfying $f(x) = 0$. These x values satisfying the constraint are called the zero roots or simply the roots of the function. Nonlinear functions can have anywhere from zero real roots to multiple roots, and generally, the solution can only be found by iterative algorithms. The iterative algorithms start with an approximate solution, the chances of finding a correct solution increases with guesses closer to a solution of the non linear function. The number of iterations to obtain the solution also depends on the initial guess.

There are multitudes of algorithms designed to be efficient depending on the characteristics of the given function. As an example, we will examine an iterative method that uses derivatives of the function: the *Newton-Raphson* algorithm. This algorithm calculates the tangent (derivative) of the function

at the current estimate of the root and uses this result to improve the estimate. The improvement is performed by calculating the intercept of the tangent with the x-axis, which represents the updated root estimate. With x_n representing the current estimate and x_{n+1} the updated estimate, an iteration step in the algorithm applies the following equation to get the updated estimate:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{1.36}$$

This method uses a loop that iterates over the estimates until the error is less than a predefined tolerance. If we want to find a root of the following function:

$$y = x^2 + x - 16 \tag{1.37}$$

we could use the following *Python* code:

```
def roots_NR(xo):
    def f_x(x):
        return x**2 + x - 16
    def fp_x(x):
        return 2*x + 1
    tol = 1e-16
    steps = 0
    while f_x(xo) > tol and steps < 20:
        xo = xo - f_x(xo)/fp_x(xo)
        steps += 1
    print(xo, steps)
roots_NR(5) #  3.53113, 5
```

A set of methods extends the use of derivatives for finding roots in nonlinear systems of equations. For vectors and vector-valued functions, the following is the linear approximation of $f(\mathbf{x})$ at $\mathbf{x}_n$:

$$f(\mathbf{x}) = f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n)(\mathbf{x} - \mathbf{x}_n) \tag{1.38}$$

where ∇ represents the gradient of the function (derivatives of the function with respect to all variables). The goal is to find $\mathbf{x}$ such that $f(\mathbf{x}) = 0$; if we choose $\mathbf{x}_{n+1}$, then:

$$f(\mathbf{x}_n) + \nabla f(\mathbf{x}_n)(\mathbf{x}_{n+1} - \mathbf{x}_n) = 0 \tag{1.39}$$

Since $\nabla f(\mathbf{x}_n)$ is a square matrix, the solution of this equation is:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \nabla f(\mathbf{x}_n)^{-1} f(\mathbf{x}_n) \tag{1.40}$$

In practice, the algorithm involves solving first the equation:

$$\nabla f(\mathbf{x}_n)\Delta\mathbf{x} = -f(\mathbf{x}_n) \tag{1.41}$$

Then, use the solution vector $\Delta\mathbf{x}$ to improve the estimate:

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \Delta\mathbf{x} \tag{1.42}$$

The *Python* code below, for example, is used to estimate the roots of the following nonlinear system:

$$x_1^3 + x_2 - 3 = 0 \tag{1.43}$$

$$-x_1 + x_2^3 + 1 = 0 \tag{1.44}$$

```
import numpy as np
def functions(x):
    return np.array([(x[0]**3 + x[1] - 3),
                     (x[1]**3 - x[0] + 1)])
def gradient(x):
    return np.array([[3 * x[0]**2, 1],
                     [-1         , 3 * x[1]**2]])
xo = np.array([0.5, 0.5])
n = 0
while abs(np.sum(np.absolute(functions(xo)))) > 1e-16 and n < 20:
    n += 1
    f = functions(xo)
    g = gradient(xo)
    delta_x = np.dot(np.linalg.inv(g), f.T)
    xo -= delta_x
print(n, xo)
```

```
20 [1.3226968  0.68590645]
```

Note the use of $\nabla f(\mathbf{x}_0)^{-1}$, which, in practice, is never used. Instead, nonlinear root finding algorithms use linear methods to solve the system in equation (1.41).

In the *Python* ecosystem, several root-finding algorithms are provided by *SciPy*. To illustrate the use of these algorithms, let's find the roots of the following nonlinear system:

$$f(x) = 3x_1 - \cos(x_2 x_3) - 0.5 \tag{1.45}$$

$$g(x) = x_1^2 - 81(x_2 + 0.1)^2 + \sin(x_3) + 1.06 \tag{1.46}$$

$$h(x) = e^{-x_1 + x_2} + 20x_3 + (10\pi - 3)/3 \tag{1.47}$$

To find the roots, we first create a *Python* function that accepts an array with the three parameters (x_1, x_2, and x_3) and returns an array with the calculated values of the three functions. Then, we call the **root** function from **scipy.optimize**, to which we provide just the function and a starting guess for the solution (it uses the default algorithm):

```
import scipy.optimize as opt
from math import cos, sin, e, pi
def non_linear_system(x):
    f = 3*x[0] - cos(x[1]*x[2])-1/2
    g = x[0]**2 - 81*(x[1]+0.1)**2 + sin(x[2]) + 1.06
    h = e**(-x[0]*x[1]) + 20*x[2] + (10*pi-3)/3
    return np.array([f, g, h])
x = np.array([0.7, 0.2, -0.3])
# uses default method = 'hybr'
solution = opt.root(non_linear_system, x)
print(solution.x)
```

```
[ 5.00000000e-01 -2.20236650e-14 -5.23598776e-01]
```

Chapter 2

Four Laws

$$dU = TdS - PdV$$

Thermodynamics deals with energy and its transformations, a concept that is implicit in the definition of the first and second law of thermodynamics. This chapter introduces the four laws of thermodynamics, along with some auxiliary functions. The laws lay the foundation for the derivation of thermodynamic potentials used to solve problems of chemical equilibrium (phase equilibrium). The chapter title is borrowed from an excellent science divulgation book by Peter Atkins, *Four Laws That Drive the Universe*.

2.1. The Zeroth Law

The zeroth law of thermodynamics introduces the concept of temperature, a property that allows us to anticipate whether there is thermal equilibrium between two chemical systems. The equilibrium between chemical systems in contact through a diathermic wall (one that allows heat exchange) can be evaluated by monitoring temperature changes. Observable temperature changes in the chemical systems indicate that they are not in thermal equilibrium. This leads to the well-known statement of the zeroth law: "If A is in thermal equilibrium with B, and B is in thermal equilibrium with C, C will be in thermal equilibrium with A."

A closed system consisting of two bodies in thermal equilibrium has a total energy (E_T):

$$E_T = E_1 + E_2 \tag{2.1}$$

In terms of E_2:

$$E_2 = E_T - E_1 \tag{2.2}$$

Thermal equilibrium means the system has a maximum value of entropy (S_T):

$$S_T = S_1(E_1) + S_2(E_2) \tag{2.3}$$

$$\frac{dS_T}{dE_1} = \frac{dS_1}{dE_1} + \frac{dS_2}{dE_2}\frac{dE_2}{dE_1} \tag{2.4}$$

(Note: entropy is explained in detail later in this chapter. For the moment think of entropy as an extensive quantity related to the tendency of heat to flow from a hotter to a colder body). From equation (2.2):

$$\frac{dE_2}{dE_1} = -1 \tag{2.5}$$

then:

$$\frac{dS_T}{dE_1} = \frac{dS_1}{dE_1} - \frac{dS_2}{dE_2} = 0 \tag{2.6}$$

$$\frac{dS_1}{dE_1} = \frac{dS_2}{dE_2} \tag{2.7}$$

Thermal equilibrium in the system leads to an equality of the derivative of the entropy of a body with respect to its energy, and this is defined as a property of the system that is reciprocal to temperature:

$$\frac{1}{T} = \frac{dS}{dE} \tag{2.8}$$

Strictly speaking, and following its statistical mechanics definition, entropy is a dimensionless quantity. The absolute temperature then has dimensions of energy (e.g., ergs). However, temperature is commonly expressed in kelvin (K), and the conversion factor between ergs and kelvin is the Boltzmann constant ($k_B = 1.3806488 \times 10^{-23} J/K$). Since temperature is usually expressed in K, the definition of entropy includes the factor k_B:

$$S = -k_B \sum_q P_q \log P_q \tag{2.9}$$

where P_q is the probability of occupation of the qth state. From a microscopic perspective, temperature is a property that depends on the relative population of energy levels in a system at equilibrium (see, for example, Angeli *et al.*, 2013). As temperature rises, populations migrate from lower to higher energy levels. At absolute zero, only the lowest state is occupied; when the temperature approaches infinity, all states are equally populated.

According to the Boltzmann distribution (for its derivation, see Engel and Reid, 2013, and Van Ness, 1983) the probability of a quantum state (P_q) with an energy E_q is given by:

$$P_q = \frac{e^{-\beta E_q}}{\sum_q e^{-\beta E_q}} = \frac{e^{-\beta E_q}}{Z} \tag{2.10}$$

where $Z = \sum_q e^{-\beta E_q}$ is known as the partition function, and $\beta = \frac{1}{k_B T}$ is a constant. As can be seen in the Boltzmann distribution equation, P_q depends solely on E_q; therefore, temperature has a statistical meaning, as it depends on the relative population of energy levels given by P_q.

Worked example 2.1

The one-particle partition function (a single-particle state) is expressed as:

$$Z_1 = 1 + e^{\frac{-\epsilon}{k_B T}} + e^{\frac{-2\epsilon}{k_B T}} + \ldots + e^{\frac{-(n-1)\epsilon}{k_B T}}$$

The N-particle partition function (independent-particle approximation) is given by:

$$Z_N = (1 + e^{\frac{-\epsilon}{k_B T}} + e^{\frac{-2\epsilon}{k_B T}} + \ldots + e^{\frac{-(n-1)\epsilon}{k_B T}})^N$$

Using these equations along with the Boltzmann distribution function, make a graph of the relative population of the energy levels (available in a system) vs. temperature. Suppose number of energy levels = 4, number of particles = 500.

```
k_B = 1.3806488e-23   # J/K -- Boltzmann constant
e = 1.602176634e-19   # charge of a electron in J
epsilon = 0.012 # eV (energy difference between levels)
levels = np.arange(0,4,1)
particles = np.arange(100,600,100)
dt = 0.01; k = 3
T = np.arange(1,1000+dt,dt)
p = np.zeros((len(T),len(levels)))
z = np.zeros((len(T),len(particles)))
for j in range(len(T)):
    zt = 0             # value of the partition function
    for i in range(len(levels)):
        p[j,i] = np.exp(-((i-1)*epsilon*e)/(k_B*T[j]))
        zt += p[j,i] # partition function for one particle
    z[j,k] = zt**particles[k] # total partition function f
    p[j,:] = p[j,:] / zt      # probability
fig, ax = plt.subplots()
ax.plot(T,p)
ax.set_xlabel(r'Temperature (K)')
ax.set_ylabel(r'Probability')
```

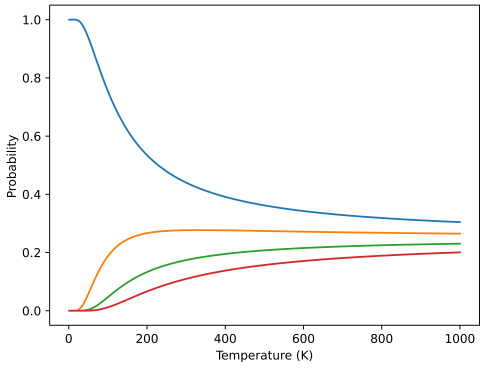

Figure 2.1: Relative population of the energy levels vs temperature.

2.2. The First Law

The simplest statement of the first law of thermodynamics is that "energy is conserved." A better definition: “There is a function of state within a system, called the internal energy, which depends only on the initial and final thermodynamic state of the system.”

With the statement of energy conservation, the first law helps clarify the meaning of the elusive concept of *internal energy*. First, energy is a function of some measurable properties, and the internal energy is simply the total energy of the system, the sum of the energies of all molecules and their interactions. Second, and following the previous statement, *internal energy* is a state function i.e., a property that only depends on the current state of the system and is independent of how it got to that state. Third, systems, regardless of how they are defined, have the capacity to do work (W), and that capacity is related to the *internal energy* (U) of the system. In a simplified way, any process driving a change in the state of a system can be described with the equation:

$$W = U(final) - U(initial) \tag{2.11}$$

However, careful observations also tell us that, during the process, temperature differences between the system and the surroundings cause energy transfer. This means the amount of work required to change the state is higher than the simplified view represented in the equation above (which is a special case, where systems change state without changes in energy due to heat, this type of system is called an adiabatic system). The new variable that takes into account the energy transfer across the system boundaries as a result of temperature differences is called heat (Q).

The full definition of the *internal energy* of a system becomes "the difference between the available heat and the useful work done by the system." In a closed and simple system (there is only mechanical work, PdV) we have:

$$U = Q - W \tag{2.12}$$

which in differential form is:

$$dU = dQ - dW \tag{2.13}$$

Work is force times distance, in terms of pressure and volume:

$$dW = F \cdot dx \left(\frac{area}{area}\right) = \left(\frac{F}{area}\right)(area \cdot dx) = P \cdot dV \tag{2.14}$$

Substituting this work expression into the equation (2.12), we have:

$$dU = dQ - PdV \tag{2.15}$$

Worked example 2.2

Suppose you have 10.0 liters of an ideal gas in an adiabatically isolated closed system, and the system is allowed to expand to a volume of 25.0 liters at a constant pressure of 3 bar. Calculate the work involved in this process and the heat transfer (Q) if conditions were isothermal (instead of adiabatic).
To find the solution, we will use the following equation:

$$W = -\int_{V_1}^{V_2} PdV = -P(V_2 - V_1)$$

Note that this is an irreversible work since expansion is done at constant P. In the second part of the problem, temperature remains constant, therefore heat must flow in the system:

$$\Delta U = Q - W = 0$$

```
P = 3 * 10**5 # Pa
V1 = 10e-3 # m3
V2 = 25e-3 # m3
W = -P *(V2-V1)      # -4.5 kJ
Q = W                # -4.5 kJ
```

2.3. Molar Heat Capacity

It can be overseen from the zeroth law that temperature is related to the energy content of a system. The quantification of this relationship is achieved by the introduction of a new thermodynamic variable, the molar heat capacity symbolized by the letter C. In formal terminology, molar heat capacity is the capacity of materials to absorb energy in the form of heat (Q). It is defined as the amount of energy absorbed by 1 mole of a substance when its temperature is raised by 1 °C:

$$C = \frac{dQ}{dT} \tag{2.16}$$

The last equation is an incomplete definition of C because dQ depends on how the change in temperature occurs. Therefore, this equation needs

then a specification of the conditions under which heat transfer is occurring. There are two conventional forms for defining C, the first is a constant volume condition

$$C_V = \left(\frac{dQ}{dT}\right)_V \tag{2.17}$$

A constant-volume reversible process implies $dU = dQ$ and the specific heat at constant volume becomes:

$$C_V = \left(\frac{dU}{dT}\right)_V \tag{2.18}$$

The second convention to define C is using a constant pressure constraint:

$$C_P = \left(\frac{dQ}{dT}\right)_P \tag{2.19}$$

The derivation of the equivalent definition of C under a constant pressure constraint in terms of a state variable is deferred until the introduction of a new state variable.

2.4. The Second Law

The second law of thermodynamics provides a basis for understanding why any change occurs at all and implies the existence of another thermodynamic property: entropy (S). While the first law introduced the concept of conservation of energy through transformations, but even with this framework in mind, the second law states that energy cannot be transformed in any way possible. This impossibility is related to entropy, which serves as a measure of the quality of energy, and the second law governs the natural increase in entropy. There are two equivalent statements of the second law:

- Kelvin's statement: No cyclic process is possible in which heat is taken from a hot source and completely converted to work.
- Clausius statement: Heat does not pass from a body with a low temperature to a body with a high temperature without a change occurring elsewhere.

A change in entropy is the ratio between the energy (in joules) transferred as heat to or from a system and the temperature (in kelvin) at which it is transferred, so its units are joules per kelvin (J/K). For the mathematical

treatment of the second law, it is useful to use the following statement of the second law: "the entropy of the universe increases in the course of any spontaneous change." In thermodynamics, spontaneous processes are those that do not have to be driven by some kind of work. The following expression of the second law, in terms of entropy, captures the Kelvin and Clausius statements:

$$dS \geq \frac{dQ}{T} \tag{2.20}$$

For a reversible process, heat transfer is given by:

$$dQ = TdS \tag{2.21}$$

then the first law equation becomes:

$$dU = TdS - PdV \tag{2.22}$$

2.5. The Third Law

The third law of thermodynamics states that the entropy of a perfect crystalline structure that has only one state with minimum energy is zero at zero kelvin. Note, however, that there is an impossibility of reaching absolute zero and entropy at zero kelvin is obtained by extrapolation; this may lead to positive values of entropy at zero kelvin which is known as *residual entropy*. Some substances may have more than one ground state; then, its *residual entropy* is determined by the number of different ground states.

The third law is used as a basis for calculating entropy at some temperature T, by combining equations (2.19) and (2.21) and taking $T_0 = 0\,K$:

$$\int_{T_0}^{T} dS = \int_{0}^{T} \frac{C_P}{T} dT \tag{2.23}$$

This is called *third law entropy*. In this equation, C_P is a function of temperature, the solution of the integral requires the knowledge of how heat capacity C_P varies with temperature.

2.6. Thermodynamic Potentials

Entropy belongs to a group of related parameters (variables) that have the property of showing the directionality of geochemical processes. These parameters are called thermodynamic potentials, with the capacity to effect

change. In the case of entropy, directionality is related to the increase of the parameter in spontaneous adiabatic processes. Because these parameters are related, if one of them is known, it is possible to derive the rest. One method for deriving these parameters is using the Legendre transform, using the definition of entropy as a starting point. This derivation is possible by application of Duhem's Law or principle: the equilibrium state in a system of fixed composition is determined by fixing two state variables at most.

2.6.1. Legendre Transform

The Legendre transform is a mathematical technique by which a particular nonlinear function is represented as a function of its derivatives. In two dimensions, the original function is represented as a collection of lines tangent to the function (where the intercept of these lines with the y-axis is the function defined as the Legendre transform).

Suppose you have a function $Y(X)$, the Legendre transform, using partial derivatives with respect to the variable X, is given by:

$$I_X = Y - \left(\frac{\partial Y}{\partial X}\right) X \tag{2.24}$$

The Legendre transform for a function with more than one independent variable, $Y(X_1, X_2, \ldots, X_n)$, is given by:

$$I_{X_1, X_2, \ldots, X_n} = Y - \sum_i \left(\frac{\partial Y}{\partial X_i}\right) X_i \tag{2.25}$$

The Legendre transform can take any combination of the independent variables (partial Legendre transforms). The geometric significance of the Legendre transform can be seen in the worked example 2.3, where the curve $y = 0.25x^2$ in a two-dimensional space is expressed in terms of a family of lines tangent to the curve.

In the following sections, enthalpy and the Gibbs free energy potentials are presented as partial Legendre transforms of U. There is another common Legendre transforms of U, the Helmholtz Free Energy that will not be discussed in this book. Additionally, an alternative set of Legendre transforms start with the entropy function; these are know as *Massieu* functions and they are particularly useful in irreversible thermodynamics.

Worked example 2.3

Make a representation of the function $y = 0.25x^2$ using a Legendre transform and plot a family of lines using the derived equation.

---

1. Find the derivative of the function:

$$\frac{\partial y}{\partial x} = 0.5x$$

2. Construct the Legendre transform:

$$I_X = y - (0.5x) * x = 0.25x^2 - 0.5x^2 = -0.25x^2$$

3. Elimination of the x variable. Replace x with the slope of the lines tangent to the curve:

$$m = 0.5x$$

$$x = 2m$$

$$I_X = 0.25(2m)^2 - 0.5(2m) * (2m) = m^2 - 2m^2 = -m^2$$

4. Find equations for a family of lines tangent to the curve, $y = mx + b$, here the intercept (b) is given by I_X (by definition):

$$y = mx - m^2$$

5. Write *Python* code to do calculations. This part serves to introduce symbolic calculations using the *Python* module *sympy* and a useful function (**sympy.lambdify()**) to make a callable function from a symbolic function.

```
import sympy as sp
x, m = sp.symbols("x m") # Define variables as symbols:
y = (1/4)*x**2           # non-linear function
dy_dx = sp.diff(y, x)    # m = slope = 0.5*x
fx = sp.lambdify(x, y)   # transform symbolic function
                         # in a python callable function
I_x = y - x * dy_dx      # Legendre transform = -0.25*x**2
x_ = sp.solve(dy_dx - m, x, dict=True)  # x in terms of m
I_m = I_x.subs(*x_)       # Legendre transform in terms of m
I_c = sp.lambdify(m, I_m) # transform symbolic function
                          # in a python callable function
x_range = np.linspace(-10, 10, 1000)    # range in x
fig, ax = plt.subplots()
# plot original function
ax.plot(x_range, fx(x_range), 'r', linewidth=2);
# plot a family of tangent lines
for i in range(7):
    tl = lambda x: (i - 3)*x + I_c((i - 3))
    ax.plot(x_range, tl(x_range), 'k--')
ax.set_ylim(-1, 20)
```

Worked example 2.3 (cont.)

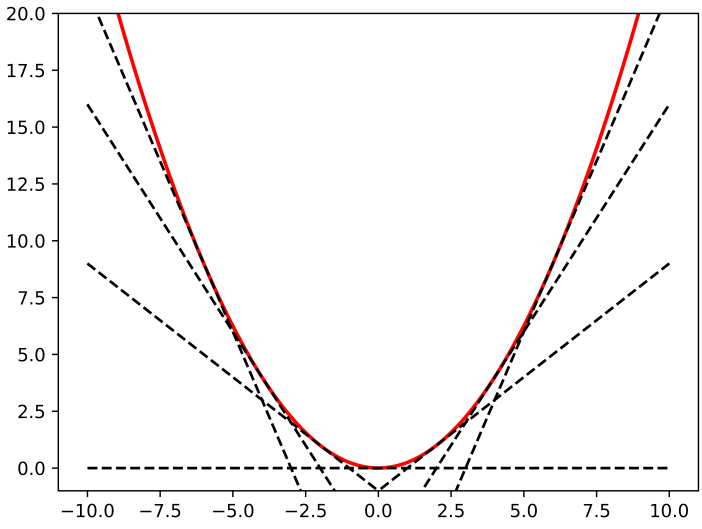

Figure 2.2: Representation of the function $y = 0.25x^2$ with a family of tangent lines using a Legendre transform.

2.7. Enthalpy

The enthalpy potential is the partial Legendre transform of of internal energy (U), where pressure (P) replaces volume (V) as an independent variable:

$$H = I_V = U - \left(\frac{\partial U}{\partial V}\right) V \tag{2.26}$$

$$H = U - (-P)\, V \tag{2.27}$$

$$H = U + PV \tag{2.28}$$

Enthalpy (H) is defined as a system variable that measures the energy released as heat by a system free to expand or contract as a process occurs. The term PV indicates the amount of work required to accommodate the gases that are generated. In other words, this energy is not available to the

surroundings in the form of heat. Differentiating equation (2.28), we get the differential expression for enthalpy:

$$dH = TdS + VdP \tag{2.29}$$

2.8. Heat Capacity at Constant Pressure

It is time to revisit the heat capacity at constant pressure defined in equation (2.19). At constant pressure, equation (2.29) becomes $dH = TdS$, and from the second law $dQ = TdS$, then:

$$C_P = \left(\frac{dH}{dT}\right)_P \tag{2.30}$$

2.9. Standard Enthalpy of Formation

The standard enthalpy of formation (ΔH_f^0) can be calculated taking into account the convention of assigning $\Delta H_f^0 = 0$ for pure elements in the standard state (gas, liquid, or solid). ΔH_f^0 is defined as the energy required to form 1 mole of a substance from the elements in the standard state. For example, the molar enthalpy of formation of quartz at 298 K and 0.1 MPa is calculated from the energy released or required as heat in the following reaction:

$$Si_{(metal)} + O_{2\,(gas)} \leftrightarrow SiO_2 \; \Delta H_f^0 = -910.648\; J/mol \tag{2.31}$$

This value of ΔH_f^0 is the molar enthalpy of formation of the substance (quartz) at 298 K, 0.1 MPa. As we saw earlier, entropy has a more universal reference state: the entropy of any pure crystalline substance at 0 K is zero. The standard data method is used to calculate ΔH_r^0 from ΔH_f^0 values of reactants and products:

$$\Delta H_r^0 = \sum n\Delta H_{f\,(products)}^0 - \sum n\Delta H_{f\,(reactants)}^0 \tag{2.32}$$

2.10. Gibbs Free Energy

The Legendre transform of U that simultaneously replaces the entropy with the temperature and the volume with the pressure as independent variables

is the Gibbs potential or Gibbs free energy.

$$G = I_{S,V} = U - \left(\frac{\partial U}{\partial S}\right) S - \left(\frac{\partial U}{\partial V}\right) V \tag{2.33}$$

$$G = U - TS + PV \tag{2.34}$$

Using the definition of H above, the mathematical expression of this function becomes:

$$G = H - TS \tag{2.35}$$

For a reversible, closed and simple system, this equation in differential form is:

$$dG = -SdT + V\,dP \tag{2.36}$$

The Gibbs free energy provides a measure of the chemical energy of a system. Since chemical systems naturally tend toward states of minimum Gibbs free energy, minimizing this parameter provides information on the stability of systems. This makes Gibbs free energy the most useful thermodynamic function in geology, as it is expressed in terms of temperature (T) and pressure (P) as characteristic variables. A more detailed discussion of G is presented in Chapter 4.

2.11. Behavior of Specific Heat Functions (Molar Heat Capacity)

Molar heat capacity is defined as the amount of heat necessary to raise the temperature by 1°C of a mole of a substance. Mathematically, this is expressed in equation (2.30); so, the specific heat is the rate of change of enthalpy with temperature. Figure 2.3 shows experimental results of C_P measurements for ferrosilite and illustrates two important points about the behavior of this function. First, at low temperatures, a peak is observed that represents a type of phase transition (known as *lambda* because of the shape of the peak) within a mineral; in this case the *lambda* transition is caused by a magnetic transition of Fe^{+2}. Second, at high temperatures, it is observed that the data follow a linear trend with a positive slope.

It should be noted that the main mechanism for heat absorption in solids is lattice vibrations (it can also be absorbed by electronic and magnetic transitions). Einstein originally developed a model describing the relationship

between C_V and lattice vibrations. This model was later modified by Debbie; the resulting equations allow us to calculate a theoretical limit of C_V at high temperatures (*Dulong-Petit limit*): $C_V \rightarrow 3nR$, where n is the total number of atoms in the solid molecule (in ferrosilite it is $3*10*R = 249.4\ J/K$). The specific heat at constant volume (equation (2.18)) is related to the specific heat at constant pressure (equation (2.19)) by the equation:

$$C_P = C_V + \alpha^2 \kappa V T \tag{2.37}$$

The equivalent of the *Dulong-Petit limit* for C_P is:

$$C_P \rightarrow 3nR + \alpha^2 \kappa V T \tag{2.38}$$

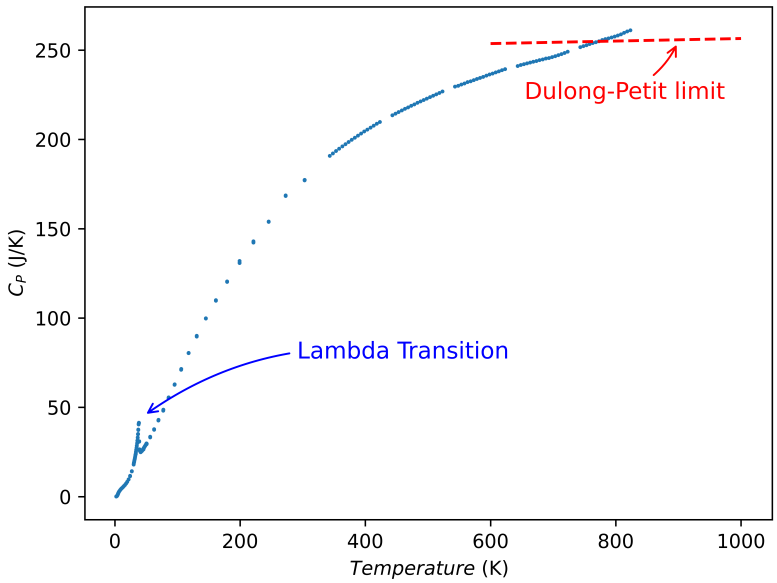

Figure 2.3: Molar heat capacity of ferrosilite showing a lambda transition and the *Dulong-Petit limit*. Data from Cemic and Dachs (2006).

The complete evaluation of the integral form of the Gibbs free energy equation—which accounts for enthalpy and entropy variations with temperature (see Chapter 4)—requires that C_P (molar heat capacity at constant pressure) be known as a function of temperature. The dependence of heat capacity on T is usually expressed as a polynomial function. The

C_P function expressed as a polynomial function is fitted with experimental data from measurements of C_P in a calorimeter at different temperatures (Figure 2.3). A well-known polynomial function used by the SUPCRT92 program is the *Maier-Kelley* function:

$$C_p = a + bT + \frac{c}{T^2} \tag{2.39}$$

Other widely used functions are those proposed by Berman and Brown (1985):

$$C_p = k_0 + k_1 T^{-0.5} + k_2 T^{-2} + k_3 T^{-3} \tag{2.40}$$

and by Robie *et al.* (1978):

$$C_p = a + k_1 bT + cT^{-2} + dT^{-1/2} + eT^2 \tag{2.41}$$

Holland (1981) proposed the elimination of the term with T^2 in the equation of Robie *et al.* (1978) to reproduce the behavior of the C_P curves at high temperatures (a limiting straight line with a positive slope):

$$C_p = a + k_1 bT + cT^{-2} + dT^{-1/2} \tag{2.42}$$

To improve the behavior of the C_P function, Holland (1981) included additional data points at high temperature, created with a reaction that produce the endmember of interest with other endmembers of well-known heat capacity functions. This exploits the additive characteristic of heat capacity (Helgeson *et al.*, 1978).

Worked example 2.4

Using the experimental data from Krupka *et al.* (1979) and Haselton and Westrum (1980) for synthetic grossular find the parameters for the equations in the polynomial C_p functions of *Maier-Kelley*, Robie *et al.* (1978), and Holland (1981). Improve the behavior of the C_p function using the polynomial from Holland (1981) including additional points at high temperature from a reaction that produce grossular from wollastonite and corundum (Wollastonite + Corundum = Grossular).

1. C_P curves fitted to synthetic grossular data. Experimental data is recovered from a saved *NumPy* array file (for creation of the *grss.npz* file see the code listing appendix). The **numpy.load** function brings the data to the *Python* session, the argument of the function is the name of the file with its relative path to the working directory. The code here creates two arrays (T and C_p) with data from both sources.

Worked example 2.4 (Cont.)

```
npzfile = np.load("data/grss.npz")
H2_T = npzfile['H2_T'], K_T = npzfile['K_T']
H2_Cp = npzfile['H2_Cp'], K_Cp = npzfile['K_Cp']
Grs_T = np.concatenate((H2_T, K_T))
Grs_Cp = np.concatenate((H2_Cp, K_Cp))
```

2. Define the polynomial functions. These are going to be used in a curve fitting algorithm from *SciPy* optimization routines:

```
def Cp(T, a,b,c,d): #HP98
    return a + b*T + c*T**(-2) + d*T**(-1/2)
def Cp_R(T, a,b,c,d,e): #Robie
    return a + b*T + c*T**(-2) + d*T**(-1/2) + e*T**(2)
def Cp_MK(T, a,b,c): #Maier-Kelley
    return a + b*T + c*T**(-2)
```

3. Find the coefficients of C_P polynomial functions using *curve_fit* from *SciPy*. Figure 2.4 shows the shape of the curves fitted to the synthetic grossular data. The determination of the coefficients in these equations is done using the least squares method.

```
from scipy.optimize import curve_fit
pars_MK, pcov = curve_fit(f=Cp_MK, xdata=Grs_T, ydata=Grs_Cp,
                  p0=[0, 0, 0], bounds=(-np.inf, np.inf))
a_MK, b_MK, c_MK  = pars_MK[0:3]
pars_R, pcov = curve_fit(f=Cp_R, xdata=Grs_T, ydata=Grs_Cp,
                  p0=[0, 0, 0, 0, 0], bounds=(-np.inf, np.inf))
a_R, b_R, c_R, d_R, e_R = pars_R[0:5]
pars, pcov = curve_fit(f=Cp, xdata=Grs_T, ydata=Grs_Cp,
                  p0=[0, 0, 0, 0], bounds=(-np.inf, np.inf))
a, b, c, d = pars[0:4]
```

4. Calculate C_p for wollastonite and corundum using known coefficients and extrapolate values for grossular (Figure 2.5), add the extrapolated to the experimental values, and find the coefficients of the polynomial function.

```
Grs_T_e = np.linspace(start=1000, stop=2000, num=20)
Wo_Cp = Cp(Grs_T_e, 0.1593, 0, -967.3, -1.0754)*1000
Cor_Cp = Cp(Grs_T_e, 0.1395, 5.89e-6, -2460.6, -0.5892)*1000
Grs_Cp_e = 3*Wo_Cp + Cor_Cp
Grs_T_HP = np.concatenate((Grs_T, Grs_T_e))
Grs_Cp_HP = np.concatenate((Grs_Cp, Grs_Cp_e))
pars, cov = curve_fit(f=Cp, xdata=Grs_T_HP, ydata=Grs_Cp_HP,
                  p0=[0, 0, 0, 0], bounds=(-np.inf, np.inf))
a2, b2, c2, d2 = pars[0:4]
```

Worked example 2.4 (cont.)

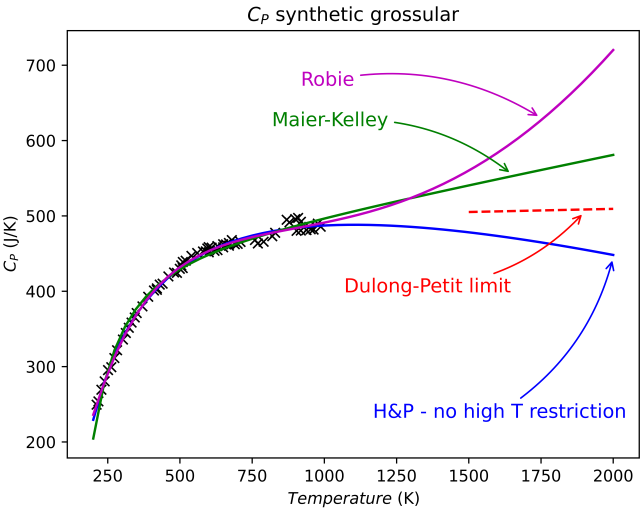

Figure 2.4: Molar heat capacity of grossular. Data from Haselton and Westrum (1980), Krupka *et al.* (1979).

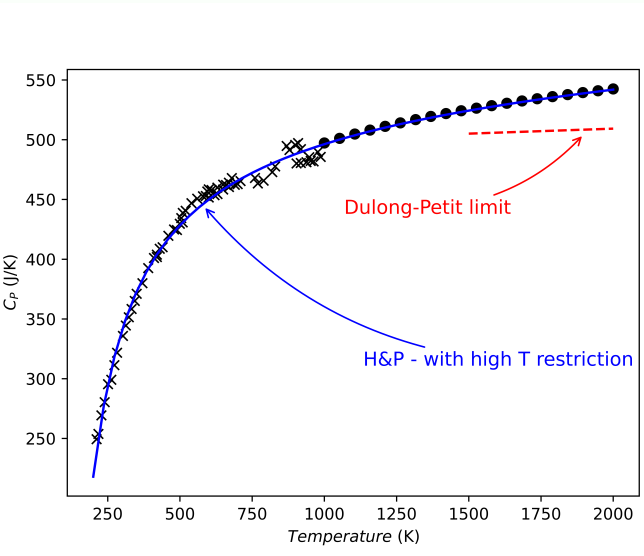

Figure 2.5: Molar heat capacity of grossular. Same as in Figure 2.4 with extrapolated data from $3Cp_{wo} + Cp_{cor} = Cp_{gr}$ at high temperature.

2.12. Thermodynamic Datasets

Thermodynamic data tables can be grouped into two broad categories: those that list the phases individually and those that relate the properties of minerals (internally consistent). An example of the compilation approach is the thermodynamic data tabulated by Robie and Hemingway (1995).

The derivation of internally consistent databases is performed using (i) regression by the least squares method, (ii) mathematical programming (linear, quadratic, and non-linear), and (iii) Bayesian techniques. All cases involve adjustable and non-adjustable quantities when producing the internally consistent database. Enthalpy and entropy are commonly used as adjustable parameters, while non-adjustable parameters include heat capacity and molar volume. The regression method does not ensure consistency with the primary data, but it provides uncertainty information. In contrast, mathematical programming ensures consistency with the primary data, but does not provide uncertainty information. For details and critical evaluation of thermodynamic datasets, see Engi (1992) and Lanari and Duesterhoeft (2019).

Examples of thermodynamic databases derived using regression by the least squares method are presented in Powell and Holland (1985), Holland and Powell (1985), Holland and Powell (1990), Holland and Powell (1998), and Holland and Powell (2011). These studies used weighted least squares regression on calorimetric, phase equilibria, and natural mineral partitioning data to determine ΔH_f^0 values for endmembers. To minimize the effect of inconsistent brackets during minimization, they used *composite data* (Powell & Holland, 1993). Their latest dataset (Holland & Powell, 2011) uses univariant and divariant equilibria to simultaneously solve for enthalpies and mixing properties of solid solutions.

The mathematical programming method is used in Berman *et al.* (1986) and Berman (1988), they used phase equilibrium constraints combined with volumetric and calorimetric data to obtain better estimates of the position of univariate reactions in $P-T$ diagrams. The system of equations to solve involve equalities and inequalities, and the problem consists of minimizing a function subject to a set of constraints. For details of the mathematical programming method, refer to Chatterjee (1991). The Bayesian method is used in Chatterjee *et al.* (1998), the technique combines the advantage of (i) and (ii), i.e., ability to handle inequalities and equalities in energy differences of reactions, the elimination of inconsistent experiments, and ability to get a variance-covariance matrix.

2.13. A Dataset Reader

In this book, we are going to work with the dataset from Holland and Powell (2011) (ds62). To set up calculations, the first step is to get thermodynamic data into a container to use within *Python* code. To accomplish this, we construct a dataset reader. To open the dataset text file, we use the *Python* function **open()**, then, we use the **readlines()** function to get an array of text lines from the file. To close the file automatically after getting the array with text lines, we use the **with** statement:

```
with open("data/tc-ds62.txt") as dsFile:
    dsContents = dsFile.readlines()
```

Note that in this example, the file tc-ds62.txt is located in a folder named "data" within the current working directory of the *Python* session.

The dataset reader must consider the structure of the dataset, which is based on blocks of thermodynamic data for endmembers (thermodynamic data of *Line 4* will be introduced in the following chapters):

- Line 1: endmember name and composition.
- Line 2: Enthalpy, entropy, and molar volume.
- Line 3: Heat capacity coefficients.
- Line 4: *EOS* (terms in equations for solids, gases, aqueous solutions, and melts) and Gibbs free energy of ordering (*Landau/BW*).

Also, at the end of the thermodynamic data for endmembers, the dataset has a big covariance matrix with errors associated with enthalpy (this matrix is not extracted in the example code).

Each line is processed to obtain the listed data using a for loop:

```
for line in dsContents:
    dsLine = line.split()
    dataArray = list(map(float, dsLine))
```

According to the structure, the reader will save the data in dictionaries. For example, for the second line in each block it uses:

```
DS_em = DS_em | {'h':dataArray[0],'s':dataArray[1],
                 'v':dataArray[2]}
```

The operator | merges existing dictionary contents with the new items. All thermodynamic endmember data is stored temporarily in a `dictionary` and converted to a *Pandas* DataFrame. The dataset can then be used within *Python* for thermodynamic calculations, as shown in the following chapters. This is the complete code listing:

```
import pandas as pd
elements = ["Si", "Ti", "Al", "Fe", "Mg", "Mn", "Ca", "Na", "K",
            "O", "H", "C", "Cl", "O2-", "Ni", "Zr", "S", "Cu",
            "Cr"]
dsContents = []
with open("data/tc-ds62.txt") as dsFile:
    dsContents = dsFile.readlines()
dsInfo = dsContents[0].split()
em_count = int(dsInfo[0])
DS = {}
for i in range(3, em_count*4, 4):
    dsLine1 = dsContents[i].split()
    dsLine2 = dsContents[i+1].split()
    dsLine3 = dsContents[i+2].split()
    dsLine4 = dsContents[i+3].split()
    em = dsLine1.pop(0)
    # Capitalize the first character
    em = em[:1].upper() + em[1:]
    comp = {}
    totalElements = 0
    for i in range(1, len(dsLine1)-1, 2):
        i_element = int(dsLine1[i])
        content = float(dsLine1[i+1])
        totalElements += content
        comp[elements[i_element-1]] = content
    DS_em = {"comp": comp}
    # H, S, V
    dataArray2 = list(map(float, dsLine2))
    DS_em = DS_em | {'H':dataArray2[0],'S':dataArray2[1],
                     'V':dataArray2[2]}
    # Cp
    dataArray3 = list(map(float, dsLine3))
    DS_em = DS_em | {'a':dataArray3[0],'b':dataArray3[1],
                     'c':dataArray3[2], 'd':dataArray3[3]}
    # EOS
    dataArray4 = list(map(float, dsLine4))
    DS_em = DS_em | {'alpha': dataArray4[0],
                     'kappa': dataArray4[1],
```

(continues on next page)

(continued from previous page)

```
                    'kappa_p': dataArray4[2],
                    'kappa_pp': dataArray4[3]}
    code = float(dataArray4[4])
    DS_em['flag'] = int(code)
    gases = ['CH4', 'H2', 'CO', 'H2S', 'S2', 'H2O', 'CO2']
    if not(em in gases or code == -1):
        DS_em['theta'] = 10636.0 / (dataArray2[1] * \
                                    1000/totalElements + 6.44)
    match code:
        case 1.0: # Landau ordering
            tc, s_max, v_max = dataArray4[5:8]
            landau = {'s_max': s_max, 'tc': tc, 'v_max': v_max}
            DS_em['ordering'] = landau
        case 2.0: # Bragg & William ordering
            h,v,wh,wv,n,fac = dataArray4[5:11]
            BW = {'h': h, 'v': v, 'wh': wh, 'wv': wv,
                  'fac': fac, 'n': n}
            DS_em['ordering'] = BW
        case -1.0: # aqueous
            DS_em['cpAq'] = float(dataArray4[5])
        case value if value != 0.0: # melt
            DS_em['flag'] = 5
            DS_em['dK'] = value
    DS[em] = DS_em
dataset = pd.DataFrame(DS)
dataset.set_index(dataset.columns[0]);
```

The following is an example of the data within the dataframe:

```
print(dataset[['Sill']]);
```

```
                                                  Sill
comp                        {'Al': 2.0, 'Si': 1.0, 'O': 5.0}
H                                              -2585.79
S                                                0.0954
V                                                 4.986
a                                                0.2802
b                                             -0.000007
c                                               -1375.7
d                                               -2.3994
alpha                                          0.000011
kappa                                            1640.0
kappa_p                                            5.06
```

(continues on next page)

(continued from previous page)

```
kappa_pp                                                    -0.0031
flag                                                              2
theta                                                    579.145113
ordering  {'h': 4.75, 'v': 0.01, 'wh': 4.75, 'wv': 0.01,...
dK                                                              NaN
cpAq                                                            NaN
```

Worked example 2.5

Use a simplified equation to calculate $\triangle G_R$ at 5 kbar and 500 °C for the polymorphic transformation $Ky = And$.

To calculate the change of Gibbs free energy in the reaction we will use the simplified equation $\triangle G_P^T = \triangle H_{T_0}^0 - T\triangle S_{T_0}^0 + P\triangle V$.

1. Get properties (H, S, and V) from the dataset:

```
data = dataset[['Ky', 'And']].iloc[1:4]
```

2. Calculate deltas:

```
ΔH = data.And.H - data.Ky.H
ΔS = data.And.S - data.Ky.S
ΔV = data.And.V - data.Ky.V
```

3. Calculate $\triangle G_R$

```
ΔGr = ΔH - (500 + 273.15) * ΔS + 5 * ΔV # 0.8820199999997271
```

The result is a positive value, this means that at the pressure and temperature conditions (5 kbar, 500 °C) G_{And} is greater than G_{Ky}. Since natural systems tend to be in the lowest energy state possible, kyanite should be the stable phase at the specified conditions.
This finding can be generalized to any reaction by saying:

1. If $\triangle G_R > 0$ reactants are stable
2. If $\triangle G_R = 0$ reactants and products are stable
3. If $\triangle G_R < 0$ products are stable

Chapter 3

Compositional and Reactive Spaces

$f = c - p + 2$

3.1. Compositional Space

3.1.1. Definition of Components

A chemical component is defined by its chemical formula, where the coefficients do not necessarily have to be rational or positive. However, the units must be related to conservative properties: (e.g., number of atoms or moles). Volume, example, is useless because it depends on other variables (e.g., P and T).

In general, the number of components considered are c linearly independent components, i.e., none of the components can be expressed by linear combinations of other components. A system with c components can be represented in a space with $c - 1$ dimensions:

- 2 components - linear space (one dimension).
- 3 components - two-dimensional space.
- 4 components - three-dimensional space.

The *compositional space* is defined as the space whose axes are the components. Take as an example a three-component system that can be represented in two-dimensional space. It is common practice to use equilateral triangles to represent the compositions of the species or phases present in such systems (Figure 3.1). Note that the region with positive values is located inside the triangle and regions outside have negative values for at least one of the components.

3.1.2. Components of a Phase

For an analysis of the compositional space, there are two important concepts to consider: (i) the number of components in which a homogeneous substance can be separated (known as endmembers) and (ii) the number of components that can vary independently. Quartz, for example, has only one endmember, meaning its composition is fixed. Plagioclase can be separated into two endmembers (i) albite and (ii) anorthite, there are plagioclase phases whose composition lie between the two extremes, but these are not considered as components. In plagioclase, however, only one component is independent.

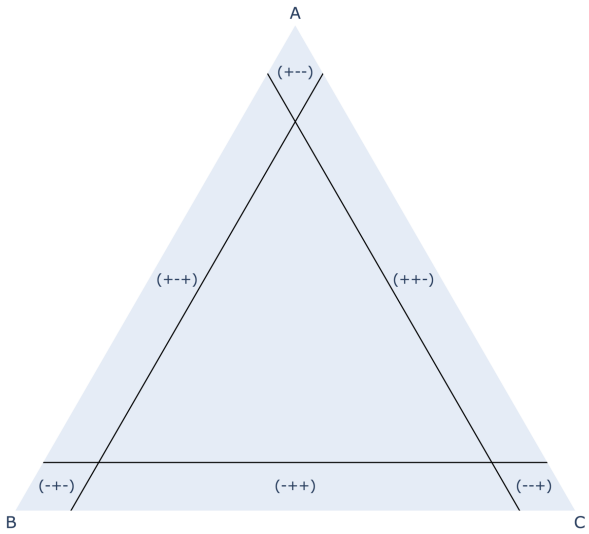

Figure 3.1: Equilateral triangle used to represent a three dimensional compositional space. Signs are in the order (a,b,c).

3.1.3. Component Transformations

To transform the description of a system from one set of components to another, we must write equations that express the content of the new components (**N**) in terms of the old ones (**O**):

$$\mathbf{O} = \mathbf{A}^{\mathrm{T}}\mathbf{N} \tag{3.1}$$

where **A** is the matrix with the coefficients of the equations that relate new and old components. The system of equations can then be solved using linear programming techniques (this is a linear transformation, as seen in Chapter 1):

$$\mathbf{N} = (\mathbf{A}^{\mathrm{T}})^{-1}\mathbf{O} \tag{3.2}$$

For this equation to work, the determinant of matrix **A** must be different from zero. When the new components have not been properly selected, the determinant can be zero, and the matrix is not invertible (there are some components that depend linearly on the others).

3.1.4. Applications

3.1.4.1. Calculations of Components (Endmembers) of a Mineral Phase

This transformation corresponds to a mapping from moles of oxides (a set of old components) to endmember proportions (the new set of components).

3.1.4.2. Calculation of Normative Rock Compositions

Calculations are analogous to calculations of endmember proportions; the difference is that the new components are not linearly independent.

3.1.4.3. Balancing Chemical Reactions

Given the composition of `m` minerals in a system with `n` components, how can these minerals be combined to form a new one? This is exactly the same problem discussed above—that is, what is the transformation that gives us the proportions of the minerals in the new coordinate system?

3.1.4.4. Projections

Projections are one of the most important applications in phase equilibria studies. They consist of reducing the number of components in a system to get a simplified representation (with loss of information). The procedure is effectively the projection or transformation of a compositional space into subcompositional spaces with fewer dimensions; therefore, it is a transformation of components. For triangular projections, the transformation matrix is constructed with the first three columns representing the vertices of the triangle, and the last $c - 3$ columns representing the projecting points, with the rows corresponding to the old components.

Worked example 3.1

Transform the analysis of a pyroxene in wt% (SiO_2 = 57.1, FeO = 6.5, MgO = 35.2, CaO = 1.0) into components as endmembers enstatite ($Mg_2Si_2O_6$), ferrosilite ($Fe_2Si_2O_6$), and diopside ($CaMgSi_2O_6$). Note that there is a pyroxene composition ($CaFeSi_2O_6$) lying withing the compositional space of the three selected endmembers.

We have four old and three new components, but the stoichiometry of pyroxene places a new restriction (it dictates that the number of cations must be four or that the sum of the new components must be 1). Ignoring SiO_2 (all endmembers have the same content for this old component) results in a square compositional matrix. The mapping of old to new components is used to construct the transformation matrix **A**:

- Enstatite - En = 2 * Mg + 0 * Fe + 0 * Ca
- Ferrosilite - Fs = 0 * Mg + 2 * Fe + 0 * Ca
- Diopside - Di = 1 * Mg + 0 * Fe + 1 * Ca

The code below starts by converting the analysis to molar proportions, then defines the compositional matrix **A**, and uses equation (3.2) to get endmember proportions:

```
import numpy as np
# Molecular weights of oxides
FeO_fw = 71.8464
MgO_fw = 40.3044
CaO_fw = 56.0794
# Conversion to molar proportions
MgO_m = 35.2/MgO_fw
FeO_m = 6.5/FeO_fw
CaO_m = 1.0/CaO_fw
molar = np.array([MgO_m, FeO_m, CaO_m])
molar = molar / np.linalg.norm(molar, 1)  # normalized to 1
#              En Fs Di
At = np.array([[2, 0, 1],  # Mg
               [0, 2, 0],  # Fe
               [0, 0, 1]]) # Ca
em = np.dot(np.linalg.inv(At), molar.T)
                              #  En   Fs   Di
em = em/np.linalg.norm(em, 1) # [0.87 0.09 0.04]
```

Worked example 3.2

Transform the analysis of a pyroxene (SiO_2 = 49.17, MgO = 8.98, FeO = 11.76, CaO = 11.48, Al_2O_3 = 15.09, Na_2O = 3.52) into components as endmembers enstatite ($Mg_2Si_2O_6$), ferrosilite ($Fe_2Si_2O_6$), diopside ($CaMgSi_2O_6$), Ca-tschermakite ($CaAlAlSiO_6$), and jadeite ($NaAlSi_2O_6$).

This example involves calculating the number of atoms for a specified number of oxygens in the structural formula of a mineral (*apfu*). This step is necessary to determine how much aluminum is allocated in tetrahedral (^{T}) and octahedral (^{M}) sites in pyroxene: $Al^T = 2 - Si$ (all Si goes to ^{T}) and $Al^M = Al - Al^T$.
(i) Divide oxides weight percent by oxides formula weight to get molar proportions (unnormalized).
(ii) Sum all molar proportion multiplied by the number of oxygens in the oxide.
(iii) Calculate an oxygen factor by dividing number of oxygens in the structural formula with the result of (ii).
(iv) To get *apfu*, each molar proportion is multiplied by the oxygen factor and by the number of cations in the oxide. The mapping of old to new components considers only Al^M and ignores Si:

- Enstatite - En = 2 * Mg + 0 * Fe + 0 * Ca + 0 * Al_6 + 0 * Na
- Ferrosilite - Fs = 0 * Mg + 2 * Fe + 0 * Ca + 0 * Al_6 + 0 * Na
- Diopside - Di = 1 * Mg + 0 * Fe + 1 * Ca + 0 * Al_6 + 0 * Na
- Ca-tschermakite - Ct = 0 * Mg + 0 * Fe + 1 * Ca + 1 * Al_6 + 0 * Na
- Jadeite - Jd = 0 * Mg + 0 * Fe + 0 * Ca + 1 * Al_6 + 1 * Na

```
import numpy as np
#                   SiO2   MgO   FeO    CaO    Al2O3   Na2O
px_anal = np.array([49.17, 8.98, 11.76, 11.48, 15.09,  3.52])
fw = np.array([ 60.08, 40.3, 71.85, 56.08, 101.96, 61.98])
no = np.array([     2,    1,     1,     1,      3,     1])
nc = np.array([     1,    1,     1,     1,      2,     2])
molar  = px_anal / fw  # molar proportions (unormalized)
sum = np.dot(no,molar) # ∑ mol prop. * oxygens in oxide
fo = 6/sum    # 2.199 # oxygen factor
Si, Mg, Fe, Ca, Al, Na = molar * nc * fo  # apfu
AlM = Al - (2 - Si)
struct_cat = np.array([Mg, Fe, Ca, AlM, Na])
#               En Fs Di Ct Jd
At = np.array([[2, 0, 1, 0, 0],  # Mg
               [0, 2, 0, 0, 0],  # Fe
               [0, 0, 1, 1, 0],  # Ca
               [0, 0, 0, 1, 1],  # Al_VI
               [0, 0, 0, 0, 1]]) # Na
em = np.dot(np.linalg.inv(At), struct_cat.T)
                              # [En   Fs   Di   Ct   Jd]
em = em/np.linalg.norm(em, 1) # [0.12 0.18 0.25 0.2  0.25]
```

Worked example 3.3

Convert back the result from the last calculation to oxide weight percent.

The transformation matrix must include all components (including Si and $Al = Al^M + Al^T$). The result is obtained by using the dot product of the transformation matrix and the endmember proportions from the last worked example and then multiplying this with the formula weight of the corresponding oxides:

```
import numpy as np
#              SiO2 MgO FeO  CaO  Al2O3 Na2O
A = np.array([[2,   2,  0,   0,   0,     0],  # En
              [2,   0,  2,   0,   0,     0],  # Fs
              [2,   1,  0,   1,   0,     0],  # Di
              [1,   0,  0,   1,   1,     0],  # Ct
              [2,   0,  0,   0, 0.5,   0.5]]) # Jd
oximol = np.dot(A.T, em.T)
oxiwt = oximol * fw
oxiwt = oxiwt/np.linalg.norm(oxiwt, 1)*100
```

Worked example 3.4

Calculate and plot the projection of anorthite ($CaAl_2Si_2O_8$), zoisite ($Ca_2Al_3Si_3O_{12}(OH)$), grossular ($Ca_3Al_2Si_3O_{12}$), and the rock composition (molar percent): SiO_2 = 76.761, Al_2O_3 = 16.772, K_2O = 2.509, CaO = 2.086, Na_2O = 1.871 into the ACN triangular plane. The projection is made from quartz (to eliminate SiO_2), H_2O, and pure muscovite (to eliminate K_2O).

The first three columns of the transformation matrix are for Al_2O_3, CaO and Na_2O (vertices of the triangle), and the last three columns for muscovite, quartz, and water (projecting points). The projection coordinates are given by equation (3.2); note that new components are located in the first three rows of the resulting matrix after applying equation (3.2).

There are various options to construct triangular plots in *Python/Jupyter*; here, we will use the *Plotly* module. For this, we use the scatter ternary option to construct a *trace* which includes circles (as markers) and text.

```
import numpy as np
At = np.array([[0,    0,  0,    3, 1, 0], # SiO2
               [1,    0,  0,  1.5, 0, 0], # Al2O3
               [0,    0,  0,  0.5, 0, 0], # K2O
               [0,    0,  0,    1, 0, 1], # H2O
               [0,    1,  0,    0, 0, 0], # CaO
               [0,    0,  1,    0, 0, 0]])# Na2O
```

Worked example 3.4 (Cont.)

```
At_inv = np.linalg.inv(At)  # inverse transformation matrix
# compositions in full system - last line is rock
X = np.array([[2,   1, 0,   0, 1, 0], # An
              [3, 1.5, 0, 0.5, 2, 0], # Zo
              [3,   1, 0,   0, 3, 0], # Grss
              [76.761, 16.772, 2.509, 0.000, 2.086, 1.871]])
X_p = np.dot(At_inv, X.T)[:3,:]
X_p = X_p / X_p.sum(axis=0)         # normalized to 1
a = X_p[0]; c = X_p[1]; n = X_p[2]  # ACN coordinates
# ---------------------------------- Triangular Diagram
import plotly.graph_objects as go
fig = go.Figure(go.Scatterternary({ 'mode':'markers+text',
           'a': a, 'b': c, 'c': n, 'marker':{ 'symbol':100,
           'color':['red','green','blue','magenta'],
           'size':10, 'line':{'width':2, 'color':'white'}},
           'text':['an', 'zo', 'grss', 'rock'],
           'textposition':'bottom right',
           'cliponaxis': False}))
fig.update_layout({'ternary':{ 'sum':1, 'aaxis':{'title':'A'},
                     'baxis':{'title':'C'},
                     'caxis':{'title':'N'}},
                     'height': 500 })
```

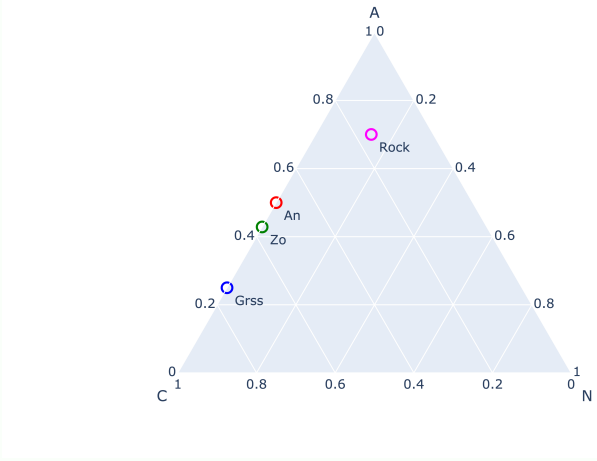

Figure 3.2: ACN projection. Projection from Ms, Qz and H_2O in the NCKASH system.

3.2. Reactive Space

3.2.1. Geometric Analysis

This is a prelude to phase equilibrium, the objective of this section is to understand the geometric analysis of univariant reactions around invariant points. For this analysis, it is important to introduce the phase rule. The phase rule relates the number of components of a system (c) and the number of phases present (p) (under specific conditions) with the variance of the system (f). Mathematically, it is expressed as:

$$f = c - p + 2 \tag{3.3}$$

The number 2 to the right of the equation represents the variables pressure and temperature. When a system of c components is analyzed, we can manipulate this equation to know the number of phases that participate in invariant points ($f = 0$) and in univariant reactions ($f = 1$):

$$p = c + 2 \quad \rightarrow \quad phases\ at\ invariant\ points \tag{3.4}$$

$$p = c + 1 \quad \rightarrow \quad phases\ in\ univariant\ reactions \tag{3.5}$$

For example, in a three-component system, there are five phases participating around an invariant point; emanating from this invariant point, there are reactions that involve four phases.

3.2.2. Number of Invariant Points and Univariant Reactions

Korzhinskii (1959) introduced the term multisystem to refer to chemical systems with more phases than can coexist in mutual equilibrium. The study of multisystem grids is carried out through the theory of groups and combinatorics to identify all possible elements in phase diagrams. For example, in pressure-temperature diagrams, univariant reaction lines ($f = 1$) intersect at invariant points ($f = 0$) and separate divariant fields ($f = 2$). Each invariant point is surrounded by $c + 2$ univariant (*non-degenerate*) reactions.

A combinatorial formula allows us to identify the number of topological elements in the diagram considering the phase rule. This is a formula in which the number of possible combinations in a set can be calculated by

taking parts of the set. In a set with X elements, the number of groupings with Y elements ($Y < X$) is given by:

$$N = \frac{X!}{Y!(X-Y)!} \tag{3.6}$$

For example, in a system where X different phases can be found (p_t), the number of invariant points is determined by applying the combinatorial formula with this number of total phases and with the number of phases that intervene in the invariant points ($Y = p_i$), a value derived from the phase rule:

$$Invariants = \frac{p_t!}{p_i!(p_t - p_i)!} \tag{3.7}$$

The invariant points represent the intersection of r_i reactions involving p_u phases. The value of r_i, the univariant reactions surrounding an invariant point, is again given by the combinatorial formula:

$$r_i = \frac{p_i!}{p_u!(p_i - p_u)!} \tag{3.8}$$

The total number of possible reactions (r_t) in the system is given by the combinatorial formula taking the total phases ($X = p_t$) and the phases involved in the univariants ($Y = p_u$):

$$r_t = \frac{p_t!}{p_u!(p_t - p_u)!} \tag{3.9}$$

However, the minimum number of independent reactions (r_{min}) in a chemical system is smaller and is equal to:

$$r_{min} = p - c \tag{3.10}$$

The argument for the derivation of this relationship is that each independent endmember has a one-to-one correspondence with one of the phases of the system. All remaining phases have compositions that can be expressed in terms of the selected independent endmembers (or corresponding phases).

Worked example 3.5

For a system with $c = 3$ and $p = 6$, determine the number of invariant points, number of reactions around each invariant, the number of phases involved in invariants and reactions, and the total number of possible reactions.

```
from math import factorial
c = 3   # Components
pt = 6  # Phases
pi = c - 0 + 2 # Invariant points have 5 phases
pu = c - 1 + 2 # Univariant reactions have 4 phases
invariants = factorial(pt)/(factorial(pi)*factorial(pt-pi)) # 6
ri = factorial(pi) / (factorial(pu)*factorial(pi-pu)) # 5
rt = factorial(pt) / (factorial(pu)*factorial(pt-pu)) # 15
```

3.2.3. Number of Components

The selection of components to describe a system may be straightforward for simple systems, but in most cases, it is not. In simple cases, the number of components can be derived by simple observation and applying the relation $c = N - r$, where N is the number of species and r the number of independent reactions that can be written between species. Two simple rules must be checked when selecting components for a system: (i) the selected set of components must be sufficient to describe the system composition and (ii) components must be independent. Consider a system with $CaCO_3(s)$, $CO_2(g)$, and $CaO(s)$. The following reaction occurs at moderate temperatures:

$$CaCO_3(s) = CO_2(g) + CaO(s) \tag{3.11}$$

By simple inspection, we can find that $N = 3$ (three phases) and $r = 1$ (one reaction); therefore, the number of components in this system is $c = 2$ (a binary system). At higher temperatures, a second reaction is possible:

$$CaO(s) = Ca(s) + 1/2O_2(g) \tag{3.12}$$

Under this new conditions, the system has one more solid phase and another gas that enters the already present gas phase, then $c = N - r = 5 - 2 = 3$ (a ternary system).

In a more general case, to determine the number of components it is necessary to construct a matrix **(A)** with rows representing composition of

the possible phases of the system in terms of all possible (identified) components. In some cases, the number of identified components equals the number of components (c), but in most cases, this will not be the case. The rank of the matrix $\mathbf{A}$ will indicate the number of components needed to describe the system.

3.2.4. Combinatorics of the Phases and Possible Reactions

The combinatorial formula provides groupings of phases that can constitute stoichiometrically valid reactions. The possible resulting reactions fall into three categories:

1. Polymorphic transformations.
2. Degenerate reactions (number of components less than the system components). These reactions have one less phase (in a ternary system, they fall along a line).
3. Reactions in the complete system.

Worked example 3.6

Determine the number of components in the *KAS* system with the phases kyanite, corundum, quartz, kalsilite, leucite, and microcline.

The compositional matrix for this system is:

$$\begin{bmatrix} Phase & Symbol & Formula \\ Kyanite & Ky & Al_2SiO_5 \\ Corundum & Cor & Al_2O_3 \\ Quartz & Qz & SiO_2 \\ Kalsilite & Kls & KAlSiO_4 \\ Leucite & Leu & KAlSi_2O_6 \\ Microcline & Mic & KAlSi_3O_8 \end{bmatrix} = \begin{bmatrix} SiO_2 & Al_2O_3 & K_2O \\ 1 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0.5 & 0.5 \\ 2 & 0.5 & 0.5 \\ 3 & 0.5 & 0.5 \end{bmatrix}$$

```
#              SiO2 Al2O3 K2O
A = np.array([[1,   1,   0], # Ky
              [0,   1,   0], # Cor
              [1,   0,   0], # Qz
              [1, 0.5, 0.5], # Kls
              [2, 0.5, 0.5], # Leu
              [3, 0.5, 0.5]])# Mic
c = np.linalg.matrix_rank(A) # components = 3
```

3.2.5. Calculation of Independent Reactions

The independent reactions can be calculated using linear algebra. For this, we formulate a relation of the form:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{3.13}$$

Where **A** is the compositional matrix, **x** is a vector with the components of the system, and **b** is the vector with the phases considered in the system. The independent reactions correspond to the last r rows of the matrix that provide the solution to the proposed equation. The algorithm consists of performing an LU decomposition of matrix **A**:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{3.14}$$

$$\mathbf{LU} \cdot \mathbf{x} = \mathbf{b} \tag{3.15}$$

In the matrix **U**, the last r rows will have zero in all their elements. Taking the matrix **L** to the right:

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{L}^{-1}\mathbf{b} \tag{3.16}$$

we obtain a matrix where the last r rows represent a set of independent reactions of the system.

Note that the LU decomposition needs to be applied to a square matrix **A** in order to make the resulting matrix **L** invertible. This is achieved by adding columns of zeros to the right of the matrix when the number of identified components is less than the number of phases.

3.2.6. Derivation of All Possible Reactions

To calculate all possible reactions, it is necessary to build compositional matrices for groups containing the maximum number of phases involved in univariant reactions (four phases). This was derived above for a three-component system with six phases. The list of reactions is obtained by applying the algorithm for independent reactions to each of the groupings.

Worked example 3.7

Find all possible combinations of phases in the *KAS* system. Also, determine the independent reactions using the compositional matrix **A** and the number of components (c) from worked example 3.6.

---

1. Combinatorics of phases. The approach followed here uses nested **for** loops. Note that *Python* has built-in functions to deal with this type of problems (this will be used in worked example 3.8).

```
from sympy import Matrix, symbols
Ky, Cor, Qz, Kls, Leu, Mic = symbols('Ky, Cor, Qz,
                                      Kls, Leu, Mic')
phases = Matrix([Ky, Cor, Qz, Kls, Leu, Mic])
phases_comb = list()
count = 0
for i in range(0, len(phases)-3):
    for j in range(i+1, len(phases)-2):
        for k in range(j+1, len(phases)-1):
            for l in range(k+1, len(phases)):
                phases_comb.append([phases[i],phases[j],
                                    phases[k],phases[l]])
print("Possible groupings (", len(phases_comb), "):")
Matrix(phases_comb)
```

There are fifteen groupings of four phases, from these groups the reactions in the system can be derived. However, many of these combinations will produce the same reaction. For example, a simple inspection of the first two groupings:

- [Ky, Cor, Qz, Kls]
- [Ky, Cor, Qz, Leu]

reveals that kalsilite and leucite cannot be involved in any reaction with the other three phases in the corresponding group, i.e., the same reaction: $Ky = Cor + Qz$ is deduced from these two groups.

2. Independent reactions.

```
import numpy as np
from scipy.linalg import lu
# add three columns of zeros to the right
A  = np.pad(A, ((0, 0), (0, 3)))
l, u = lu(A, permute_l = True)
# last 6-c rows = independent reactions
r = Matrix(np.linalg.inv(l)[c:,:])*phases
```

This produces:

- $Qz + 0.5Kls = 0.5Mic$
- $Leu = 0.5Kls + 0.5Mic$
- $Ky + 0.5Kls = Cor + 0.5Mic$

Worked example 3.8

Write a script to find all possible reactions in the *KAS* system.

The approach we will follow here for the combinatorics uses the *itertools* module of *Python*. The rest of the script is similar to worked example 3.7.

```
import numpy as np
from scipy.linalg import lu
from sympy import Matrix, symbols
import itertools
Ky, Cor, Qz, Kls, Leu, Mic = symbols('Ky, Cor, Qz,
                                     Kls, Leu, Mic')
phases = Matrix([Ky, Cor, Qz, Kls, Leu, Mic])
A = np.array([[1,   1,   0], # Ky
              [0,   1,   0], # Cor
              [1,   0,   0], # Qz
              [1, 0.5, 0.5], # Kls
              [2, 0.5, 0.5], # Leu
              [3, 0.5, 0.5]])# Mic
for i in itertools.combinations(range(6), 4):
    Ai = A[np.ix_(i, [0,1,2])]
    Ai = np.pad(Ai, ((0, 0), (0, 1)))
    c = np.linalg.matrix_rank(Ai)
    if c < 6:
        l, u = lu(Ai, permute_l = True)
        phases_i = phases[i,:]
        ri = Matrix(np.linalg.inv(l)[c:,:])*phases_i
        print(ri)
```

The above code will produce 16 reactions from 15 groups, but many of them are not unique. There are only 8 unique stoichiometrically possible reactions in this system:

- $Kls + Qz = Leu$
- $0.5\ Kls + Qz = 0.5\ Mic$
- $Leu + Qz = Mic$
- $0.5\ Kls + 0.5\ Mic = Leu$
- $Kls + Ky = Cor + leu$
- $0.5\ Kls + ky = Cor + 0.5\ Mic$
- $Ky + Leu = Mic + Cor$
- $Cor + Qz = Ky$

3.2.7. Schreinemakers

Schreinemakers's analysis (F.A.H. Schreinemakers) is used to determine the relative stability of univariant reactions in phase diagrams. This analysis is important because, when constructing a phase diagram, the stability of reaction curves in the diagram is generally unknown. In reaction curves emanating from an invariant point, there is one less phase than in the invariant. The curves are then labeled with the phase (written between square brackets) that is absent in the reaction. This notation is used in the diagram of Figure 3.3, which corresponds to what is commonly known as the triple point of the Al_2SiO_5 system (a one-component system with three phases).

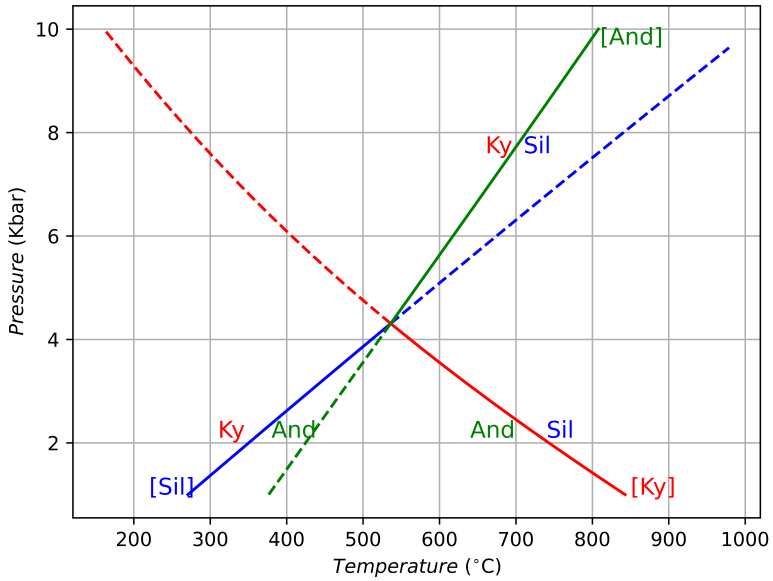

Figure 3.3: Al_2SiO_5 triple point. Individual reactions are labelled with the absent phase in the reaction.

3.2.7.1. The First Two Schreinemakers Rules

The first is the metastable extension rule, which states that the metastable extension of a reaction falls within a region where the missing phase is produced by reactions limiting the region. In the Al_2SiO_5 triple point diagram, we see that the metastable extensions of reactions $[And]$, $[Sil]$, and $[Ky]$ are found in the regions where And, Sil, and Ky are produced, respectively.

The second rule is the 180° rule, and establishes that divariant associations can only be stable in regions limited by curves that form angles of less than or equal to 180° at the intersections.

These rules are better understood with examples. Figure 3.4 is an example of a $P-T$ sketch of the topology of a binary system with four phases. This diagram has the following characteristics:

- The four mineral phases occur at an invariant point.
- Four univariant curves radiate from the invariant point, each of them is characterized by the absence of one of the phases that occurs in the invariant.
- The univariant lines divide the diagram into four divariant fields, each defined by having an association that only occurs in that field.
- Divariant fields have from zero to two metastable extensions of the univariant reaction lines.

In the field with the assemblage $A + D$, the angle between reactions $[B]$ and $[C]$ can only be increased up to the metastable extension of reaction $[C]$. Consequently, this angle is always less than or equal to 180°. The same applies for reactions that limit the other fields.

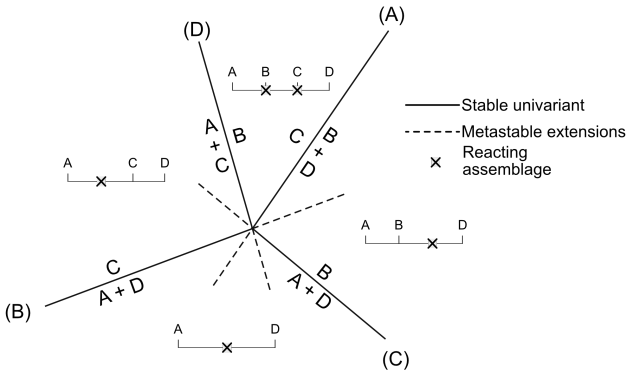

Figure 3.4: Schreinemakers analysis in a binary system.

Figure 3.5 is an example of a $P-T$ diagram for a ternary system with five phases. This example corresponds to the *AFM* system, which results from projecting the *KFMASH* system from H_2O, muscovite, and quartz. In this diagram, the five phases coexist at an invariant point, from which five univariant reactions radiate. This diagram is also characterized by divariant fields with zero to two metastable extensions. The diagram of Figure

3.5 was calculated with version 3.4 of thermocalc (Powell *et al.*, 1998) and with the database ds62 (Holland & Powell, 2011). The resulting reactions (unbalanced) are:

```
[Ky]: Grt + Ms + Chl = Bt + St + Qtz + H2O
[Grt]: St + Ms + Chl = Bt + Ky + Qtz + H2O
[Chl]: St + Ms + Qtz = Grt + Bt + Ky + H2O
[St]: Grt + Ms + Chl = Bt + Ky + Qtz + H2O
[Bt]: St + Qtz = Grt + Chl + Ky + H2O
```

In Figure 3.5, only the compatibility diagram between reactions [*Ky*] and [*Bt*] was calculated (at 600 °C and 10 kbar). The other triangular diagrams were not calculated and are only schematic representations of the equilibrium that occurs in the divariant fields when crossing the reaction lines. Note that the [*Chl*] and [*Bt*] reactions are different from those presented in the *KFMASH* grid of Wei and Powell (2003) and Wei *et al.* (2004), where the reactions are of the form:

```
[Chl]: St + Bt = Grt + Ky
[Bt]: St + Chl = Grt + Ky
```

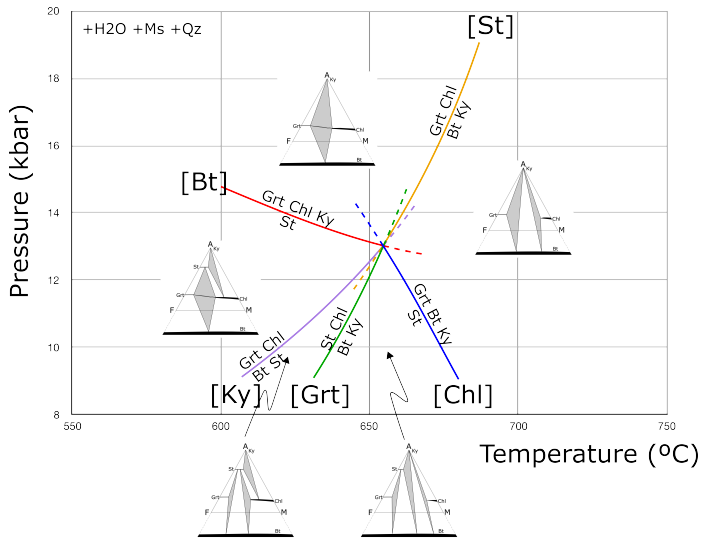

Figure 3.5: Example of a region in a $P - T$ grid with triangular compatibility diagrams in the *AFM* system projected from the *KFMASH*.

3.2.7.2. How to Orient the Diagram

Schreinemakers analysis helps to determine the relative stability; however, it tells us nothing about the orientation of the diagram in pressure-temperature ($P-T$) space. To properly draw the reaction grid, several complementary techniques can be used (depending on the information available):

- Naturally observed equilibria: natural occurrences of phases might be restricted to high or low pressures (or temperatures).
- Experimental data that provide information on the position of invariant points and univariant reaction lines in $P-T$ space.
- Thermodynamic databases that allow to determine the slope of the reaction lines in $P-T$ space (Clapeyron's equation).
- In reactions that release H_2O (or CO_2), the fluid must be on the high -temperature side of the reaction. Generally, dehydration reactions have high positive slopes in $P-T$ space.
- In solid-solid reactions, the densest phases (lower molar volume) occur on the high-pressure side of the reaction.

3.2.7.3. The *KAS* System as an Example of Schreinemakers Analysis

We have previously determined that in the *KAS* system, there are six possible invariant points ([*Kls*], [*Cor*], [*Ky*], [*Leu*], [*Mic*], and [*Qz*]) and eight stoichiometrically possible univariant reactions:

```
Kls + Qz = Leu                  [Ky, Cor, Mic]
0.5 Kls + Qz = 0.5 Mic          [Ky, Cor, Leu]
Leu + Qz = Mic                  [Ky, Cor, Kls]
0.5 Kls + 0.5 Mic = Leu         [Ky, Cor, Qz]
Kls + Ky = Cor + leu            [Mic, Qz]
0.5 Kls + ky = Cor + 0.5 mic    [Leu, Qz]
Ky + Leu = Mic + Cor            [Kls, Qz]
Cor + Qz = Ky                   [Kls, Leu, Mic]
```

Figure 3.6 is a triangular compatibility diagram with the six phases considered in the system, phases are joined by compatibility lines. The goal is to construct a diagram with all possible univariant reactions for this system and draw compatibility diagrams in the divariant regions of the diagram.

$P - T$ petrogenetic grids and compatibility diagrams are further explained in chapter six.

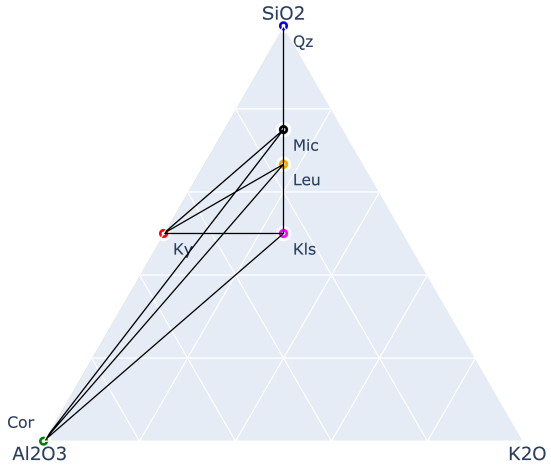

Figure 3.6: Triangular *KAS* compatibility diagram with kyanite, corundum, quartz, kalsilite, leucite, and microcline.

3.2.7.3.1 Graphical Derivation of Reactions. In compatibility diagrams, there are three possibilities for reaction topologies (Figure 3.7):

- Four phases in a quadrilateral with crossing tie lines joining the corners of the quadrilateral, reactions would be in the form $a + b = c + d$.
- Three phases in a degenerate system, all phases lie along a line, reactions are in the form $a + b = c$.
- Three phases forming a triangle and a fourth phase in the center, tie lines join the center phase with phases in the corners of the triangle. Reactions are in the form $a = b + c + d$.

The previously determined reactions in this system can also be derived by applying the topologic analysis to Figure 3.6. For example, on the left edge of the triangle, the tie line between Cor and Qz can be used to derive the reaction $Cor + Qz = Ky$; a second degenerate system is the tie line

between Kls and Qz; along this line, four reactions can be derived. There are also three quadrilaterals, each one representing a reaction in the system.

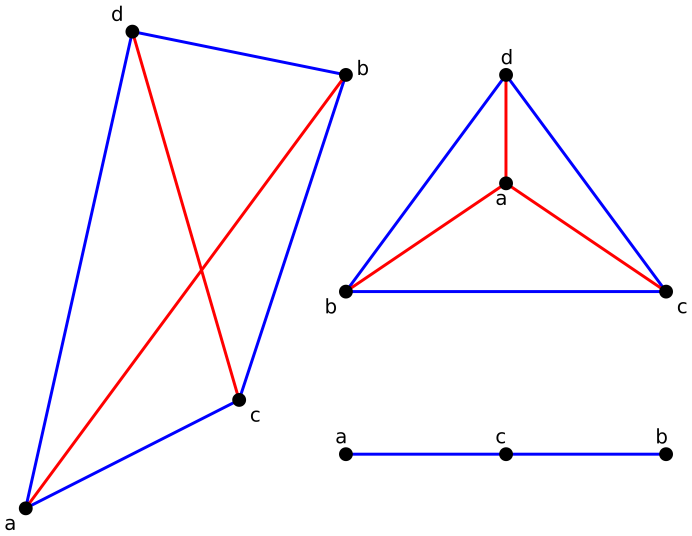

Figure 3.7: Expected topologies in triangular compatibility diagrams. The quadrilateral and the triangle are full-system reactions, and the line represents a degenerate system reaction. In the full-system topologies, compatibility (red lines) can be used to derive the reactions.

3.2.7.3.2 Grid Construction. The following analysis starts with invariant points in subsystems (which correspond to reactions in the complete system). When there are degenerate reactions, the number of reactions around the invariant will be less than the number calculated with the combinatorial formula.

- Invariant point 1 [Ky, Cor] (subsystem)

In the listed reactions above, whenever Ky is absent, so is Cor, the invariants [Cor] and [Ky] then correspond to the same point. Four reactions emanate from this point. Since this is a degenerate subsystem (fewer than the maximum possible number of phases are present), these reactions are going to remain stable when they cross other invariants (see below).

The reactions around the invariant point are:

```
Kls + Qz = Leu [Mic]
0.5 Kls + Qz = 0.5 Mic [Leu]
Leu + Qz = Mic [Kls]
0.5 Kls + 0.5 Mic = Leu [Qz]
```

Figure 3.8 shows a schematic, non-oriented diagram with the interpretation of stable and metastable extensions, interpreted according to the metastable extension rule. This diagram is derived as follows. First, choose a starting reaction and decide on which side of the reaction curve are going to be reactants and products, then draw compatibility diagrams accordingly. The choice does not matter at this point, but it is possible to get a mirror image of the correct topology. The second step involves selecting the reactions located next to the chosen reaction (left or right) and continue the procedure with all other reactions.

Let us start with the $[Mic]$ reaction with $Qz + Kls$ on the left and Leu on the right. The next step is to select the reaction that is going to be located to the right; it is clear by looking at the compatibility diagram that the next reaction should involve Qz and Leu, this is the $[Kls]$ reaction. The remaining two reactions can be added with a similar argumentation.

After having a logical sequence of reactions in the schematic diagram, the final step is to apply the metastable extension rule. Note that Leu is produced to the right of $[Mic]$ and to the left of $[Kls]$ and $[Qz]$; the metastable extension of $[Leu]$ must lie then between $[Mic]$ and $[Kls]$. The final diagram, shown in Figure 3.8, emerges after applying the metastable extension rule to the other reactions.

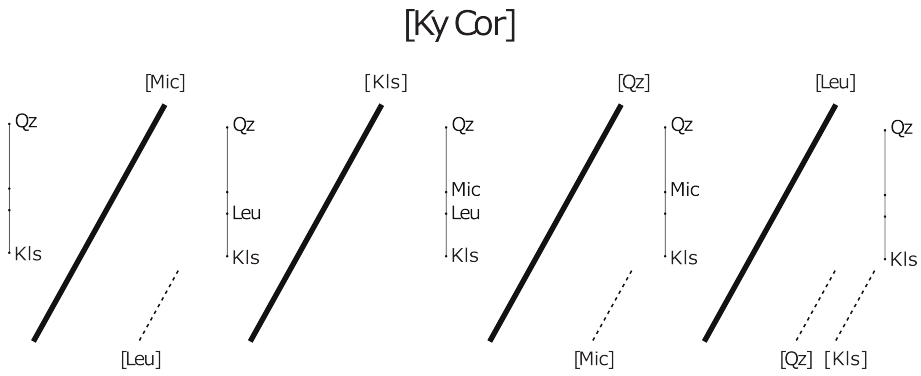

Figure 3.8: Topology of reactions around `[Ky Cor]` invariant point in the *KAS* system.

- Invariant point 2 $[Qz]$

Four reactions emerge from this invariant: three in the complete system and a degenerate reaction that comes from the invariant point [Ky, Cor]. This is the degenerate reaction [Ky, Cor, Qz], and does not change its stability when crossing the invariant point [Qz]. The reactions are as follows:

```
Kls + Ky = Cor + leu [Mic]
0.5 Kls + Ky = Cor + 0.5 mic [Leu]
Ky + Leu = Mic + Cor [Kls]
0.5 Kls + 0.5 Mic = Leu [Ky, Cor]
```

In the construction of the diagram (Figure 3.9), if we start with the [Leu] reaction (a complete system reaction), the other two reactions of the complete system need Leu. Therefore, it is necessary to locate a reaction that produces Leu on both sides of the [Leu] reaction. The candidate is the degenerate reaction [Ky, Cor]; remember that the stability of this reaction does not change as it crosses the invariant.

Alternatively, if we start with the [Kls] reaction, the [Mic] reaction can be located next to [Kls], since there is a stability line between Leu and Cor (products in the reaction as written). The [Leu] reaction cannot be next to [Kls], since using the compatibility diagrams on each side of [Kls], there is no way of forming the quadrilateral $Kls - Mic - Ky - Cor$. This confirms that the [Ky, Cor] reaction must be on the other side of [Kls].

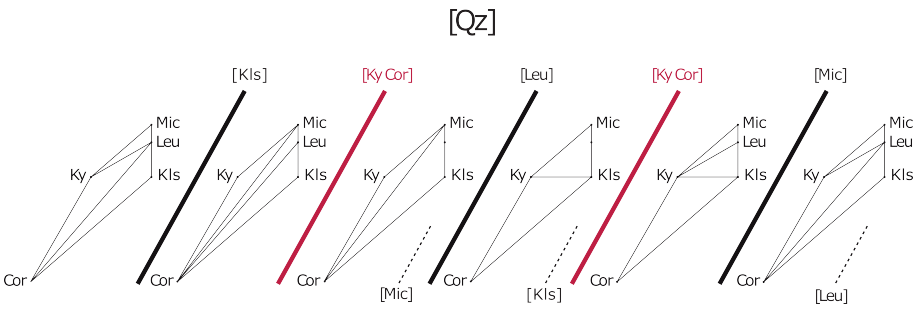

Figure 3.9: Topology of reactions around [Qz] invariant point in the *KAS* system.

- Invariant point 3 [Kls]

Three reactions emanate from [Kls], only one in the complete system and two degenerate reactions, one that comes from the invariant point [Ky, Cor], and, as previously established, these reactions do not change in stability when crossing the invariant point.

```
Ky + Leu = Mic + Cor [Qz]
Cor + Qz = Ky [Leu, Mic]
Leu + Qz = Mic [Ky, Cor]
```

We start constructing the diagram with the $[Qz]$ reaction (complete system) and continue with the two degenerate reactions on each side of $[Qz]$. Note that in these cases, Qz will appear by consumption of Mic (reaction $[Ky, Cor]$) or Ky (reaction $[Leu, Mic]$). The complete diagram is shown in Figure 3.10.

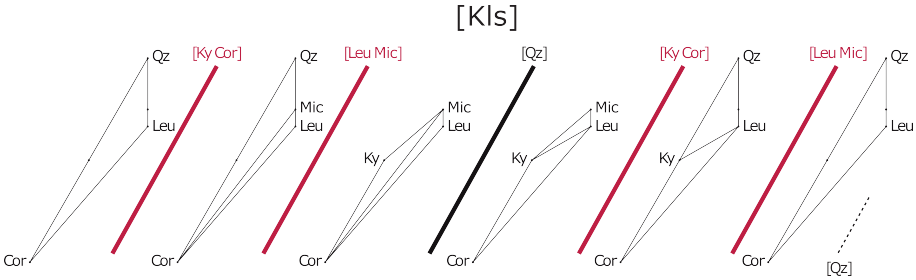

Figure 3.10: Topology of reactions around $[Kls]$ invariant point in the *KAS* system.

- Invariant points 4 $[Leu]$ and 5 $[Mic]$

Three reactions emanate from each of the invariants, only one in the complete system and two degenerate reactions. The analysis is similar to that of the invariant point $[Kls]$. The reactions for these invariant points are listed below:

$[Leu]$

```
0.5 Kls + Ky = Cor + 0.5 mic [Qz]
Cor + Qz = Ky [Mic Kls]
0.5 Kls + Qz = 0.5 Mic [Ky, Cor]
```

$[Mic]$

```
Kls + Ky = Cor + leu [Qz]
Kls + Qz = Leu [Ky, Cor]
Cor + Qz = Ky [Leu, Kls]
```

3.2.7.3.3 KAS Petrogenetic Grid. Figure 3.11 shows the pressure vs. temperature diagram illustrating the topology analyzed in the previous sections. The locations of these reactions were calculated with the thermodynamic

database ds62 (Holland & Powell, 2011) and *Python* functions to calculate Gibbs free energy and a root-finding algorithm (see Chapter 4). The invariants are labeled with the absent phase, while reactants and products. Invariants 4 and 5 are metastable; of these, only invariant 4 appears in the region of the graph. The two metastable invariants involve intersections of metastable extensions of reactions. Note also the crossing of some reaction curves that are no invariants, this is the effect of projecting reactions in the $P - T$ plane.

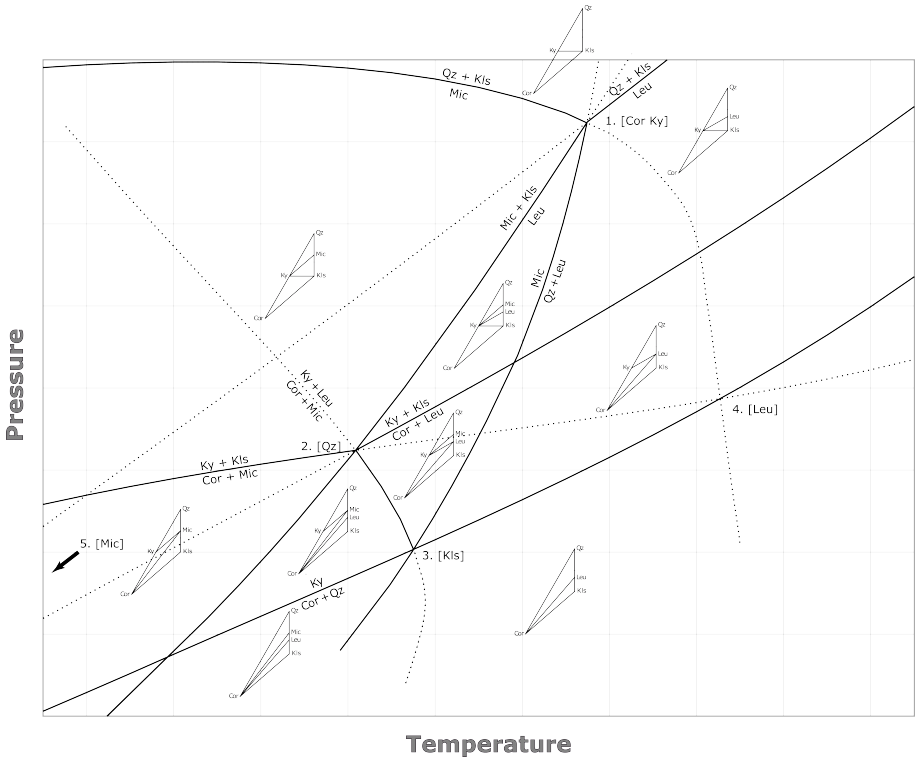

Figure 3.11: Part of a simple *KAS* petrogenetic grid with triangular compatibility diagrams in the divariant fields. The topology of this grid was derived first with a Schreinemakers analysis.

3.2.8. Summary of Chemographic Analysis in Triangular Diagrams for Systems with Pure Phases

These diagrams are useful for studying all possible univariant reactions in a simplified system and allow one to qualitatively and quantitatively locate the

univariant reactions in $P - T$ space. Additionally, they aid in understanding the evolution of systems where interactions between solid phases and aqueous solutions occur. To sum up, the following steps outline a complete chemographic analysis:

1. Define the system by determining the mineral phases of interest and choose the components of the system.
2. Specify system pressure and temperature, assuming pure phases.
3. Assume that H_2O is in excess.
4. List the compositions of the mineral phases.
5. Locate the mineral compositions within the ternary space.
6. Connect stable phases with compatibility lines.
7. Triangular regions show stable mineral associations.
8. Simplify chemography and solve problems. Start with points with several phases present. Write reactions between pairs of mineral phases, determine $\Delta_r G$, and remove unstable phases.Then, resolve pseudobinaries. Finally, resolve stability of crossing compatibility lines.

Chapter 4

Phase Equilibria in Systems with Pure Phases

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

$dG = -SdT + Vdp \quad dG = -SdT + Vdp \quad dG = -Sd$

$dT + Vdp \quad dG = -SdT + Vdp \quad dG = -SdT + Vdp$

4.1. Gibbs Free Energy (G)

Gibbs free energy is a thermodynamic potential that provides a measure of the chemical energy of a system. The differential form of the Gibbs free energy equation was derived in Chapter 2. For reference, this is the equation:

$$dG = -SdT + VdP \tag{4.1}$$

4.1.1. Gibbs Free Energy of Formation at the Reference State

The Gibbs free energy data is tabulated as "free energy of formation" data (ΔG_f^0), which is the energy change associated with the formation of one mole of a substance from its standard elements at the $P-T$ conditions of the reference state. Some tables list instead other thermodynamic data from which ΔG_f^0 can be derived.

4.1.2. Gibbs Free Energy Change in a Reaction

When analyzing free energy changes in reactions (ΔG_r), they are referred to constant pressure and temperature. The value of this change at standard state is calculated using ΔG_f^0 values of reactants and products:

$$\Delta_r G^0 = \sum n\Delta G^0_{f\ (products)} - \sum n\Delta G^0_{f\ (reactants)} \tag{4.2}$$

For reactions involving only pure phases (not solid solutions), the free energy change between reactants and products at any $P-T$ condition is expressed by using the equation (with the Gibbs free energy equation in its integral form):

$$\triangle_r G_P^T = \triangle_r H^0_{T_0} - T\triangle_r S^0_{T_0} + \int_{T_0}^{T} \triangle_r C_P dT - T\int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT + \int_{P_0}^{P} \triangle_r V dP \tag{4.3}$$

Where

$$C_P = a + bT + cT^{-2} + dT^{-0.5} \tag{4.4}$$

The subscripts and superscripts 0, P_0, and T_0 indicate standard state, the reference pressure, and the reference temperature. Note, however, that this

equation does not consider the Gibbs free energy contributions of a pure phase undergoing internal ordering.

4.1.3. Reaction Curves

If reactants and products coexist under equilibrium conditions, the Gibbs free energy change of the reaction must be zero, i.e., the pressure and temperature of this equilibrium must lie on a curve (for example, in $P-T$ space) representing the reaction. This curve is called the *reaction curve*. Mathematically, the equilibrium condition is represented by:

$$\triangle_r G_P^T = 0 \tag{4.5}$$

Assuming volume independent of pressure, we can solve for P in terms of T using equation (4.3):

$$P = -\frac{\triangle_r H_{T_0}^0 + \int_{T_0}^{T} \triangle_r C_P dT}{\triangle_r V} + \frac{\triangle_r S_{T_0}^0 + \int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT}{\triangle_r V} * T \tag{4.6}$$

$$\int_{T_0}^{T} \triangle_r C_P dT = T\triangle_r a + \frac{T^2 \triangle_r b}{2} - \frac{\triangle_r c}{T} + 2.0T^{0.5}\triangle_r d \Bigg|_{T_0}^{T} \tag{4.7}$$

$$\int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT = \triangle_r a \log(T) + T\triangle_r b - \frac{0.5\triangle_r c}{T^{2.0}} - \frac{2.0\triangle_r d}{T^{0.5}} \Bigg|_{T_0}^{T} \tag{4.8}$$

As a first approximation to the location of reaction curves in a pressure-temperature diagram, we can ignore the integrals of C_p, in other words, we assume enthalpy and entropy are independent of temperature. Then, equation (4.6) becomes:

$$P = -\frac{\triangle_r H_{T_0}^0}{\triangle_r V} + \frac{\triangle_r S_{T_0}^0}{\triangle_r V} * T \tag{4.9}$$

This equation has the form $y = mx + b$, where $m = \frac{\triangle_r S_{T_0}^0}{\triangle_r V}$ is the slope of the linearized reaction curve. The equation of the slope is known as the *Clapeyron equation*.

Worked example 4.1

Construct the kyanite-sillimanite-andalusite triple point graph using the Clapeyron equation, second term in (4.9). Assume molar volume independent of pressure and ignore the fourth and fifth terms of (4.3). Use also $T_{triple\,point}$ = 536.3 °C, $P_{triple\,point}$ = 4.31 kbar. Extract data from dataset 62.

---

1. Extract data from the dataset (this assumes you have loaded the dataset with the code from chapter 2) and calculate deltas:

```
als_ds = dataset[['Ky', 'Sill','And']].iloc[2:4]

ΔV1 = als_ds.And.V - als_ds.Sill.V
ΔV2 = als_ds.And.V - als_ds.Ky.V
ΔV3 = als_ds.Ky.V - als_ds.Sill.V
ΔS1 = als_ds.And.S - als_ds.Sill.S
ΔS2 = als_ds.And.S - als_ds.Ky.S
ΔS3 = als_ds.Ky.S - als_ds.Sill.S
```

2. Calculate slopes and intersections:

```
# triple point
Ti = 536.3 # + 273.15
Pi = 4.31
# Slopes
m1 = ΔS1/ΔV1
m2 = ΔS2/ΔV2
m3 = ΔS3/ΔV3
# Intersections
b1 = Pi - m1 * Ti
b2 = Pi - m2 * Ti
b3 = Pi - m3 * Ti
```

3. Plot triple point graph (see Figure 3.3):

```
fig, ax = plt.subplots()
ax.plot(T, m1*T+b1, 'b--', T, m2*T+b2, 'r--',
        T, m3*T+b3, 'y--')
ax.legend(('And=Sil', 'And=Ky','Sil=Ky'))
fig.show;
```

The simplification made in worked example 4.1 is valid in a few cases, but it is more of a rapid representation of reaction curves in $P - T$ space. In a more general case, it is necessary to use the complete equation (4.3).

However, solving this equation for pressure and temperature is not possible in a straight way, and it is useful to use another simplification (not as simple as the one above). The assumption is that the integral terms involving C_p are constants, i.e., they are calculated at a specific temperature and used at all other temperature conditions. This approach is called linearization of the curve at specific pressures and temperatures. Graphically, the linearized equation represents a tangent line to the reaction curve at the selected temperature.

Worked example 4.2

Construct a linearized version (at $T = 573.15\,K$) of the univariant reaction curve using equations (4.6), (4.7), and (4.8) for the reaction $Ab + Ne = 2\ Jd$ ($NaAlSi_3O_8 + NaAlSiO_4 = 2\ NaAlSi_2O_6$).

---

1. Extract thermodynamic data and calculate the value of $\int_{T_0}^{T} \triangle C_P dT$ and $\int_{T_0}^{T} \frac{\triangle C_P}{T} dT$ with $T_0 = 298\,K$ and $T = 573.15\,K$. This is a linearization of the curve at $T = 573\,K$:

```
anj_ds = dataset[['Ab', 'Ne','Jd']].iloc[1:8]
# Reaction Deltas
ΔH = 2*anj_ds.Jd.H - anj_ds.Ne.H - anj_ds.Ab.H
ΔS = 2*anj_ds.Jd.S - anj_ds.Ne.S - anj_ds.Ab.S
ΔV = 2*anj_ds.Jd.V - anj_ds.Ne.V - anj_ds.Ab.V
Δa = 2*anj_ds.Jd.a - anj_ds.Ne.a - anj_ds.Ab.a
Δb = 2*anj_ds.Jd.b - anj_ds.Ne.b - anj_ds.Ab.b
Δc = 2*anj_ds.Jd.c - anj_ds.Ne.c - anj_ds.Ab.c
Δd = 2*anj_ds.Jd.d - anj_ds.Ne.d - anj_ds.Ab.d
To = 298.15
T_l = 573.15 # Temperature of linearization (K)
# Integrals
Int_Cp = Δa*T_l + Δb/2*T_l**2 - Δc/T_l + 2*Δd*T_l**(0.5) \
         - (Δa*To + Δb/2*To**2 - Δc/To + 2*Δd*To**(0.5))
Int_CpT = (Δa*np.log(T_l) + Δb*T_l - Δc/(2*T_l**2) \
           - 2*Δd/(T_l**0.5) - (Δa*np.log(To) + Δb*To \
           - Δc/(2*To**2) - 2*Δd/(To**0.5)))
```

2. Use these values in the equation for P and graph the curve on a P vs T diagram:

```
intercept = -(ΔH+Int_Cp) / ΔV # Intercept
slope = (ΔS+Int_CpT) / ΔV      # Slope
T = np.arange(200., 620., 20.)
P = intercept + slope * T      # linearized equation
```

Worked example 4.2 (cont.)

```
fig, ax = plt.subplots()
ax.plot(T, intercept + slope * (T+273.15), 'b--')
ax.axis([200, 600, 3, 11])
ax.set_xlabel("T ($^{\circ}$C)")
ax.set_ylabel("P (kbar)")
```

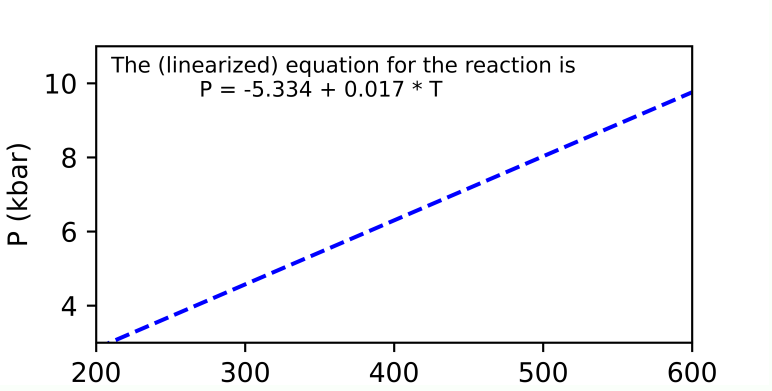

Figure 4.1: Pressure-temperature diagram with the univariant reaction curve for $Ab + Ne = 2\ Jd$ linearized at $T = 573.15\,K$ (using ds62).

4.2. Equations of State (*EOS*)

An equation of state is an expression that relates the state variables V, P, and T of a substance. The simplest example is the ideal gas equation of state:

$$PV = nRT \tag{4.10}$$

Equations of state are necessary to evaluate the Gibbs free energy of a substance at high pressure by integration of the term $V\,dp$ at constant temperature, because V is a function of pressure at the temperature of interest. The simplest non-ideal *EOS* is the *van der Waals* equation for gases:

$$P = \frac{nRT}{V - n\,b_{vdw}} - \frac{n^2\,a_{vdw}}{V^2} \tag{4.11}$$

where n is the number of moles and a_{vdw} and b_{vdw} are constants.

4.2.1. *EOS* for Gases

In the previous section, the *van der Waals* equation was introduced. However, this equation does not adequately reproduce the behavior of non-ideal gases, especially at high pressures. The *van der Waals* equation is the basis for a two-parameter *EOS* known as the *Redlich-Kwong* equation:

$$P = \frac{RT}{V_m - b_{rk}} - \frac{a_{rk}}{V_m(V_m + b_{rk})\sqrt{T}} \tag{4.12}$$

The use of this equation in the geological literature includes the treatment of the terms a_{rk} and b_{rk} as functions of P and T; these equations are known as the *modified Redlich-Kwong equation* (*MRK*). Holland and Powell (2011) database uses a version known as the *compensated Redlich-Kwong equation* (*CORK*) for the gases CO, CH_4, H_2, S_2, and H_2S. In this version, the term b_{rk} is kept constant to facilitate the analytical integration of the equation, which causes a divergence in the volume above a pressure threshold P_0. The *CORK* equation takes the form:

$$V_m = V_m^{MRK} + V_m^{Virial} \tag{4.13}$$

where the term V_m^{Virial} is an extra contribution to volume. The name of this term comes from the *Virial* type *EOS*, which expresses the compressibility Z as a function of P, such that $Z = 1$ as $P \to 0$:

$$Z(P,T) = 1 + a_v P + b_v P^2 + c_v P^3 + \ldots \tag{4.14}$$

In practice, the *CORK* equation uses the *RK* equation; solving for V:

$$V_m = \frac{RT}{P} + b_{rk} - \frac{a_{rk}(V_m - b_{rk})}{PV_m(V_m + b_{rk})\sqrt{T}} \tag{4.15}$$

with the approximation $V_m \approx RT/P$:

$$V_m \approx \frac{RT}{P} + b_{rk} - \frac{a_{rk}R\sqrt{T}}{(RT + b_{rk}P)(RT + 2b_{rk}P)} \tag{4.16}$$

two terms are added to improve *Virial* behavior:

$$V_m \approx \frac{RT}{P} + b_{rk} - \frac{a_{rk}R\sqrt{T}}{(RT + b_{rk}P)(RT + 2b_{rk}P)} + c_{rk}\sqrt{P} + d_{rk}P \tag{4.17}$$

Where:

$$a_{rk} = a_0 \frac{T_c^{5/2}}{P_c} + a_1 \frac{T_c^{3/2}}{P_c} T \tag{4.18}$$

$$b_{rk} = b_0 \frac{T_c}{P_c} \tag{4.19}$$

$$c_{rk} = c_0 \frac{T_c}{P_c^{3/2}} + \frac{c_1}{P_c^{3/2}} T \tag{4.20}$$

$$d_{rk} = d_0 \frac{T_c}{P_c^2} + \frac{d_1}{P_c^2} T \tag{4.21}$$

Table 4.1 lists the values for the coefficients (Holland & Powell, 1991) (units: a_{rk} in $kJ^2kbar^{-1}K^{1/2}mol^{-2}$, b_{rk} in $kJkbar^{-1}mol^{-1}$). Table 4.2 lists the critical temperature and pressure for gases; data for CO, CH_4, and H_2 from Holland and Powell (1991), and data for H_2S from Reid *et al.* (1987).

Table 4.1: Coefficients for the *CORK* equation

a_{rk}		b_{rk}	
a_0	a_1	b_0	
5.45963e-5	-8.63920e-6	9.18301e-4	
c_{rk}		d_{rk}	
c_0	c_1	d_0	d_1
-3.30558e-5	2.30524e-6	6.93054e-7	-8.38293e-8

Table 4.2: T_c and P_c for the *CORK* equation

	T_c (K)	P_c (kbar)
CH_4	190.6	0.0460
H_2	41.2	0.0211
CO	132.9	0.0350
H_2S	373.2	0.0882

The *EOS* used for H_2O and CO_2 in Holland and Powell (2011) is also a *Virial* type *EOS* (Pitzer & Sterner, 1994):

$$\begin{aligned} P/RT =& \rho + c_1\rho^2 - \rho^2 \left(\frac{c_3 + 2c_4\rho + 3c_5\rho^2 + 4c_6\rho^3}{\left(c_2 + c_3\rho + c_4\rho^2 + c_5\rho^3 + c_6\rho^4\right)^2} \right) \\ &+ c_7\rho^2 e^{-c_8\rho} + c_9\rho^2 e^{-c_{10}\rho} \end{aligned} \tag{4.22}$$

The molar volume is (cm^3, 1 cm^3 = 0.1 J/bar):

$$V_m = \frac{1}{\rho} \tag{4.23}$$

The parameters c_1 - c_{10} are temperature dependent:

$$c_i = c_{i,1}T^{-4} + c_{i,2}T^{-2} + c_{i,3}T^{-1} + c_{i,4} + c_{i,5}T + c_{i,6}T^2 \tag{4.24}$$

This equation needs to be solved to find V using a root-search algorithm, with the matrix of coefficients $C_{i,j}$ for CO_2 and H_2O (Table 4.3 and Table 4.4).

Table 4.3: Matrix of coefficients $C_{i,j}$ for CO_2

i,1	i,2	i,3
0	0	1.8261340e+06
0	0	0
0	0	0
0	0	-1.3270279
0	0	1.2456776e-01
0	0	0
-3.9344644e+11	9.0918237e+07	4.2776716e+05
0	0	4.0282608e+02
0	2.2995650e+07	-7.8971817e+04
0	0	9.5029765e+04

i,4	i,5	i,6
7.9224365e+01	0	0
6.6560660e-05	5.7152798e-06	3.0222363e-10
5.9957845e-03	7.1669631e-05	6.2416103e-09
-1.5210731e-01	5.3654244e-04	-7.1115142e-08
4.9045367e+00	9.8220560e-03	5.5962121e-06
7.5522299e-01	0	0
-2.2347856e+01	0	0
1.1971627e+02	0	0
-6.3376456e+01	0	0
1.8038071e+01	0	0

Table 4.4: Matrix of coefficients $C_{i,j}$ for H_2O

i,1	i,2	i,3
0	0	2.4657688e+05
0	0	5.8638965e-01
0	0	-6.2783840e+00
0	0	0
0	0	5.6654978e+03
0	0	0
3.8878656e+12	-1.3494878e+8	3.0916564e+05
0	0	-6.5537898e+04
-1.4182435e+13	1.8165390e+8	-1.9769068e+05
0	0	9.2093375e+04

i,4	i,5	i,6
5.1359951e+01	0	0
-2.8646939e-03	3.1375577e-05	0
1.4791599e-02	3.5779579e-04	1.5432925e-08
-4.2719875e-01	-1.6325155e-05	0
-1.6580167e+01	7.6560762e-02	0
1.0917883e-01	0	0
7.5591105e+00	0	0
1.8810675e+02	0	0
-2.3530318e+01	0	0
1.2246777e+02	0	0

Note that Pitzer and Sterner (1994) use:

- Only two or three terms are used; in c_6 only the fourth term is used.
- In c_1 and $c_7 - c_{10}$ there are only coefficients for negative powers of T.

- For $c_2 - c_6$ the terms with positive powers for T dominate at high T.

4.2.2. *EOS* for Solid Phases

The most commonly used equations for solids at high pressures are the *Birch-Murnaghan* type *EOS* (*B-M*), which are based on the relationship between volume and compression. In these equations, P is expressed as a polynomial function of strain ($f = -\varepsilon$). For example, the second-order *EOS B-M* is obtained by truncation of the Helmholtz free energy after the second power:

$$P = \frac{3\kappa_0}{2}\left[\left(\frac{\rho}{\rho_0}\right)^{7/3} - \left(\frac{\rho}{\rho_0}\right)^{5/3}\right] \tag{4.25}$$

where κ_0 is the isothermal compressibility modulus (or bulk modulus) at $P = 0$. In the Holland and Powell (1998) database, the second order *EOS B-M* is expressed as:

$$P = \frac{\kappa}{\kappa'}\left[\left(\frac{V^0}{V}\right)^{\kappa'} - 1\right] \tag{4.26}$$

where $\kappa'$is the pressure derivative of the compressibility modulus. The treatment of the data with this equation involves the use of a thermal expansion expression independent of the selected *EOS*, and the expansion and compression terms are related through a linear dependence of the compressibility modulus.

The *Tait* equation of state is a semi-empirical *B-M* type *EOS*, originally proposed for water that was later applied to compressible solids. However, calculations with this equation produce negative volumes at high pressures. Macdonald (1966) presented a modified *Tait* equation of state applicable to both solids and liquids, where calculations only lead to zero volumes at infinite reduced pressures ($p = P - P_0$), this equation has the form:

$$\frac{V}{V_0} = \left(1 + \frac{np}{\kappa_0}\right)^{-1/n} \tag{4.27}$$

where n is a pressure-independent parameter. The modified *Tait EOS* of Huang and Chow (1974) is presented in Freund and Ingalls (1989) as:

$$\frac{V}{V_0} = 1 - a\left[1 - (1 + bP)^{-c}\right] \tag{4.28}$$

where a, b, and c are constant parameters derived from the isothermal compressibility modulus and its pressure derivatives. The Holland and Powell

(2011) updated version of their database uses a more general *EOS*, with the implicit inclusion of a thermal expansion term based on the modified *Tait* equation of Huang and Chow (1974), called *TEOS* (*Tait-modified EOS*). In this equation, the adjustment for high temperatures is performed by adding a "thermal pressure" term (P_{th}):

$$V = V_o(1 - a_{eos}(1 - (1 + b_{eos} * (P - P_{th}))^{-c_{eos}})) \tag{4.29}$$

$$\int V dP = PV_o(1 - a_{eos} - (\frac{a_{eos}(Pb_{eos} - P_{th}b_{eos} + 1)^{1-c_{eos}}}{b_{eos}P(c_{eos} - 1)})) \tag{4.30}$$

where:

$$a_{eos} = \frac{(1 + \kappa_o')}{(1 + \kappa_o' + \kappa_o * \kappa_o'')} \tag{4.31}$$

$$b_{eos} = \frac{\kappa_o'}{\kappa_o} - \frac{\kappa_o''}{(1 + \kappa_o')} \tag{4.32}$$

$$c_{eos} = \frac{1 + \kappa_o' + \kappa_o * \kappa_o''}{\kappa_o'^2 + \kappa_o' - \kappa_o * \kappa_o''} \tag{4.33}$$

$$P_{th} = \alpha_o \kappa_o \frac{\theta}{\xi_o}(\frac{1}{e^u - 1} - \frac{1}{e^{u_o} - 1}) \tag{4.34}$$

$$\xi_o = \frac{u_o^2 e^{u_o}}{(e^{u_o} - 1)^2} \tag{4.35}$$

$$u_o = \frac{\theta}{T_o} \tag{4.36}$$

$$u = \frac{\theta}{T} \tag{4.37}$$

$$\theta = \frac{10636}{(S_o/n + 6.44)} \tag{4.38}$$

With ξ_o representing the Einstein function, which considers $\alpha \cdot \kappa$ decreasing to zero as the temperature decreases, and θ is the Einstein temperature (the number n in the definition of this temperature is the number of atoms in the phase).

The coefficient of thermal expansion is implicitly contained within the *EOS*, and its value can be calculated with:

$$\alpha = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_P\Big|_{P_0} = \alpha_0 \frac{\xi}{\xi_o}\left(\frac{1}{(1 - b_{eos}P_{th})(a_{eos} + (1 - a_{eos})(1 - b_{eos}P_{th})^{-c_{eos}})}\right) \tag{4.39}$$

The compressibility module can be calculated with:

$$\begin{aligned}\kappa &= -V\left(\frac{\partial P}{\partial V}\right)_T \\ &= \kappa_0\left(1+b_{eos}(P-P_{th})\right)\left(a_{eos}+(1-a_{eos})(1+b_{eos}(P-P_{th}))^{-c_{eos}}\right)\end{aligned} \tag{4.40}$$

4.2.3. *EOS* for Melts

Several of the *EOS* derived for solids are also used for melts. However, in the *TEOS* equation, the expression for thermal expansion is not appropriate for melts due to its foundation in vibrational models (Holland & Powell, 2011). Instead, a constant thermal expansion and a linear dependence with temperature for the compressibility module are used for melts, as in Holland and Powell (1998):

$$V = V_t(1-a_{eos}(1-(1+b_{eos}*P)^{-c_{eos}})) \tag{4.41}$$

$$\int V\,dP = V_tP - V_ta_{eos}P - V_ta_{eos}\frac{(Pb_{eos}+1)^{-c_{eos}+1}}{b_{eos}\,(c_{eos}-1)} \tag{4.42}$$

$$\int V\,dP = V_tP\left(1-a_{eos}-a_{eos}\frac{(b_{eos}P+1)^{-c_{eos}+1}}{b_{eos}P\,(c_{eos}-1)}\right) \tag{4.43}$$

Note the similarity of this equation with the function for solid phases. For this equation, we have that V_t is (equation 4 of Lange, 1997):

$$V_t = V_oe^{\alpha(T-T_o)} \tag{4.44}$$

The equation for the compressibility modulus is:

$$\kappa = \kappa_o(1-0.00015(T-T_o)) \tag{4.45}$$

4.2.4. Gibbs Free Energy and *EOS* for Aqueous Solutions

The *EOS* for aqueous solutions have been derived from empirical, macroscopic, microscopic, and electrostatic relationships, or combinations of those relationships. The *EOS* of *Helgeson-Kirkham-Flowers* is the most accepted equation, thanks to its extensive database foundation (Dolej, 2013). In this model, the specific heat capacity and the molar volume incorporate empirical contributions not related to solvation.

The Holland and Powell (2011) database uses density models for aqueous solutions. These models are based on calorimetric and volumetric properties as a function of temperature and density of the solvent. They use a logarithmic linear empirical relationship between the equilibrium constant and the density of the solvent. An example of such an expression, reduced to three parameters, is given by (Anderson *et al.*, 1991; Mesmer *et al.*, 1988):

$$\ln K = a_o + \frac{a_1}{T} + \frac{b_1}{T} \ln \rho_w \tag{4.46}$$

In Anderson (1995), the density model is taken directly into the Gibbs free energy equation to give:

$$G_i^o = \Delta_f H_i^o - TS_i^o + \frac{C_{p,i}^0}{298.15 \left(\frac{\partial \alpha}{\partial T}\right)_P^o} \left(\alpha^o (T - 298.15) + \frac{T}{T'} ln \frac{\rho}{\rho^o}\right) \tag{4.47}$$

Holland and Powell (1998) used a revised Anderson's density model with the following adjustments:

- Volume is included.
- Addition of a heat capacity term (linear with temperature: $C_p = C_p^* + b_i T$).
- Addition of a term that corrects density for temperature (at $T > 500$ K, the term T'/T).
- Correction of density model contribution for molar volumes of aqueous species at reference conditions $-C_p^* \beta^o P / (298.15 (\partial \alpha / \partial T)_P^0)$:

$$\begin{aligned} G_i^o =& \Delta_f H_i^o - TS_i^o + PV_i^o + b_i \left(298.15T - \frac{298.15^2}{2} - \frac{T^2}{2}\right) \\ &+ \frac{C_{p,i}^*}{298.15 \left(\frac{\partial \alpha}{\partial T}\right)_P^o} \left(\alpha^o (T - 298.15) - \beta^o P + \frac{T}{T'} ln \frac{\rho}{\rho^o}\right) \end{aligned} \tag{4.48}$$

where:

$$C_{p,i}^* = C_{p,i}^o - 298.15 b_i \tag{4.49}$$

and the molar volume is given by:

$$V_i = V_i^o - \frac{C_{p,i}^* \beta^0}{298.15 \left(\frac{\partial \alpha}{\partial T}\right)_P^o} \left(1 - \frac{T}{T'}\right) \tag{4.50}$$

In these expressions, V_i^0 is the standard molar volume of the species i, β^0 and α^0 are the compressibility and the coefficient of thermal expansion for H_2O at reference conditions, and b_i is a Maier-Kelley coefficient for aqueous species (heat capacity). The values for the constants are: $\beta^0 = 45.23e-6\,bar^{-1}$, $\alpha^0 = 25.93e-5\,K^{-1}$, and $(\partial\alpha/\partial T)_P^0 = 9.5714e-6\,K^{-2}$, these values are taken from Anderson *et al.* (1991).

4.3. Gibbs Free Energy of Ordering (Pure Phases)

The organization of atoms in a crystal structure can be visualized in terms of unit cells (e.g., face-centered cubic, *fcc*). In these structures, some atoms have preferences for occupying particular sites (long-range order) and for the type of other atoms that are in nearby sites in the structure (short-range order). Thus, in complete long-range ordered structures the probability of finding an atom in one of the available crystallographic sites is one, this probability decreases as the ordering decreases. Structures with complete long-range order must have also short-range order, but completely long-range disordered structures can have any degree of short-range order. In short, long-range order accounts for the distribution of atoms in crystallographic sites averaged over the complete crystal, whereas short-range order is originated by the occupancy of individual sites over small scales (a few Å).

The preferences that originated the "ordering" in a structure are related to the interaction energies of pairs of atoms. Disordered states have higher entropies and enthalpies than ordered states. However, according to the Gibbs free energy differential equation ($\partial G = \partial H - T\partial S$), at high temperatures the structures prefer the disordered state (the second term increases more), and at low temperatures the ordered state. This last analysis can be summarized in a competition between atomic interactions and configurational entropy.

In minerals with fixed composition, atoms can also have preferences for the available crystallographic sites, leading to ordering within the structure. In sillimanite, for example, Al and Si can be ordered in the available tetrahedral sites (Holland & Powell, 1996). In general, two types of ordering behavior are observed: (i) a convergent ordering, where there is a decrease in symmetry in a discrete phase transformation (from high to low temperature) and (ii) a non-convergent, ordering where the crystallographic sites can never become related by symmetry and there is no phase transition.

As a consequence of the order-disorder behavior, the Gibbs free energy of a phase changes according to the ordering state, and a term in the Gibbs integral equation takes this change into account:

$$G_P^T = H_{T_0}^0 - T\,S_{T_0}^0 + \int_{T_0}^{T} C_P dT - T\int_{T_0}^{T} \frac{C_P}{T} dT + \int_{P_0}^{P} V\,dP + G_{ord} \qquad (4.51)$$

The central idea of the models describing ordering is the introduction of a term known as the ordering parameter (Q), which describes the ordering state at different temperatures. G_{ord} is expressed in terms of this parameter, and the equilibrium of the system is determined by minimizing the Gibbs free energy equation with respect to the parameter Q. The parameter gives the probability of occupation of the sites at the given temperature, in mathematical terms, for two atoms (A and B) distributed between two crystallographic sites (s and s'), it is defined as:

$$Q = |X_A^s - X_A^{s'}| = |X_B^s - X_B^{s'}| \qquad (4.52)$$

This equation describes the atomic arrangement of atoms A and B across sites s and s', where X_A^s represents the mole fraction of A atoms at s site. From this equation, it can be seen that $Q = 0$ for the completely disordered state and $Q = 1$ for a completely ordered state of a crystal with equal proportions of A and B.

An important aspect of the order parameter Q is that its behavior around the transition temperature serves as a basis for classifying phase transitions. For example, Q changes continuously around the critical temperature for systems undergoing second-order phase transition, whereas the change is discontinuous for first-order phase transitions.

There are two macroscopic approaches to characterize the Gibbs free energy of ordering. The first one is based on the theory of phase transitions introduced by Landau in 1937, and the second is known as the point approximation (Bragg-Williams model).

4.3.1. Landau

The formalism used to study ordering in this model is based on the Landau free energy expansion (Carpenter *et al.*, 1994). This approach provides useful descriptions for displacive and order-disorder phase transformations and nonconvergent ordering (no phase transition). In Landau theory, G_{ord}

is calculated using a Taylor series expansion in the parameter Q (Holland & Powell, 1996), incorporating only even powers for Q:

$$\Delta H_{ord} = -HQ - \frac{1}{2}aT_cQ^2 + \frac{1}{4}bQ^4 + \frac{1}{6}cQ^6 + \ldots \tag{4.53}$$

$$\Delta S_{ord} = \frac{1}{2}aQ^2 \tag{4.54}$$

where the H term allows for non-convergent ordering (there is no phase transition). Combining these equations, we get the expression for G_{ord}:

$$G_{ord} = \frac{1}{2}(a(T - T_c) + a_vP)Q^2 + \frac{1}{6}aT_cQ^6 \tag{4.55}$$

In this equation, $T_c = T_c^0 - \frac{a_v}{a}P$ is the pressure-dependent critical temperature, where T_c^0 is the critical temperature at $P = 0$. From the previous equation, we can derive maximum entropy S_{ord} and maximum volume V_{ord} of disorder:

$$V_{ord} = V_{max}|_{Q=1} = \frac{1}{2}a_vQ^2 \tag{4.56}$$

$$S_{ord} = S_{max}|_{Q=1} = -\frac{1}{2}aQ^2 \tag{4.57}$$

Properties tabulated in the Holland and Powell (2011) database are for the (low temperature) ordered phase at the standard state conditions; therefore, the final expression of the Gibbs free energy of ordering is:

$$\begin{aligned} G_{ord} =& S_{max}T_c^o(Q_o^2 - \frac{1}{3}Q_o^6) - S_{max}(T_cQ^2 - \frac{1}{3}T_c^oQ^6) \\ & - T(S_{max}(Q_o^2 - Q^2)) + P(V_{max}Q_o^2) \end{aligned} \tag{4.58}$$

where:

$$T_c = T_c^o + \frac{V_{max}}{S_{max}}P \tag{4.59}$$

$$Q_o^4 = \frac{T_c^o - T_o}{T_c^o} \tag{4.60}$$

$$Q^4 = \frac{T_c - T}{T_c^o} \tag{4.61}$$

4.3.2. Bragg-Williams

The idea of the Bragg-Williams approximation (Bragg & Williams, 1934; Bragg & Williams, 1935; Williams, 1935) is based on the concept of long-range correlation. This means that the energy of a particular atom in a structure is determined by the average degree of order prevalent in the whole structure. In theory, it is assumed that local fluctuations do not affect this prevalent order, and the order parameter is a function of average distribution of atoms in the available crystallographic sites (Carpenter, 1992). As with the Landau model, the enthalpy and entropy of ordering are expressed as functions of the (long-range) order parameter; entropy is treated as being purely configurational (see below), and enthalpy is treated as arising from nearest-neighbour-type atomic interactions. In applying this formalism (Holland & Powell, 1996), the Gibbs free energy of ordering is given by:

$$\Delta G_{ord} = \Delta H_{ord} - T\Delta S_{ord} \tag{4.62}$$

where ΔS_{ord} can be derived from the general configurational entropy equation:

$$-R\left(X_A^S log X_A^S + X_B^S log X_B^S + X_A^{S'} log X_A^{S'} + X_B^{S'} log X_B^{S'}\right) \tag{4.63}$$

In the general case, where the atoms are mixed in *1:n* ratios at $n+1$ sites (Holland & Powell, 1996):

$$X_A^s = \frac{1+nQ}{n+1} \tag{4.64}$$

$$X_A^{s'} = \frac{1-Q}{n+1} \tag{4.65}$$

$$X_B^s = \frac{n-nQ}{n+1} \tag{4.66}$$

$$X_B^{s'} = \frac{n+Q}{n+1} \tag{4.67}$$

The entropy of ordering is given by:

$$S_{ord} = -facR(X_A^S log X_A^S + X_B^S log X_B^S + X_A^{S'} log (X_A^{S'})^n + X_B^{S'} log (X_B^{S'})^n) \tag{4.68}$$

and the enthalpy of ordering is given by:

$$\Delta H_{ord} = H + PV - Q^2 (PWv + Wh) + Q (-H + P (-V + Wv) + Wh) \tag{4.69}$$

The critical temperature is given by:

$$T_c = \frac{2(W_h + W_v P)}{R}(1+n) \tag{4.70}$$

In a plot of ΔG_{ord} vs Q the curve will have a minimum that corresponds to the order parameter (and the site occupancy) equilibrium value.

Worked example 4.3

Plot ΔG_{ord} vs Q for sillimanite at 5 kbar and 1000 °C.

We start by defining a function that calculates ΔG_{ord}, sillimanite ordering is described using the Bragg-Williams model:

```
def Gord(Q,P,T,n,H,V,Wh,Wv,fac):
    R = 0.0083145 #8.31446261815324
    T = T + 273.15
    Xa_s = (1.0+n*Q)/(n+1.0)
    Xa_sp = (1.0-Q)/(n+1.0)
    Xb_s = (n-n*Q)/(n+1.0)
    Xb_sp = (n+Q)/(n+1.0)
    Sord = -R*fac*(Xa_s*np.log(Xa_s) + Xb_s*np.log(Xb_s) + \
           n*Xa_sp*np.log(Xa_sp) + n*Xb_sp*np.log(Xb_sp))
    Hord = H + Q * (Wh-H) - Q**2 * Wh
    Vord = V + Q * (Wv-V) - Q**2 * Wv
    Gord = Hord - T * Sord + P * Vord
    return Gord
```

Next, extract sillimanite data from the dataset, call the function with sillimanite parameters and plot the result:

```
sil_BW = dataset.Sill.ordering
h = sil_BW['h']; v = sil_BW['v']; wh = sil_BW['wh']
wv = sil_BW['wv']; fac = sil_BW['fac']; n = sil_BW['n']
Q1 = np.arange(0., 0.99999, 0.001)
fig, ax = plt.subplots()
ax.plot(Q1, Gord(Q1,5,1000+273.15,n,h,v,wh,wv,fac))
ax.set_ylabel('G ord', fontsize=14)
ax.set_xlabel('Q', fontsize=14)
ax.set_xlim(0,1)
```

The Q parameter for the minimum ΔG_{ord} is around 0.95. In the next section, we will find a way to determine the exact value for this parameter at the minimum of Figure 4.2.

Worked example 4.3 (cont.)

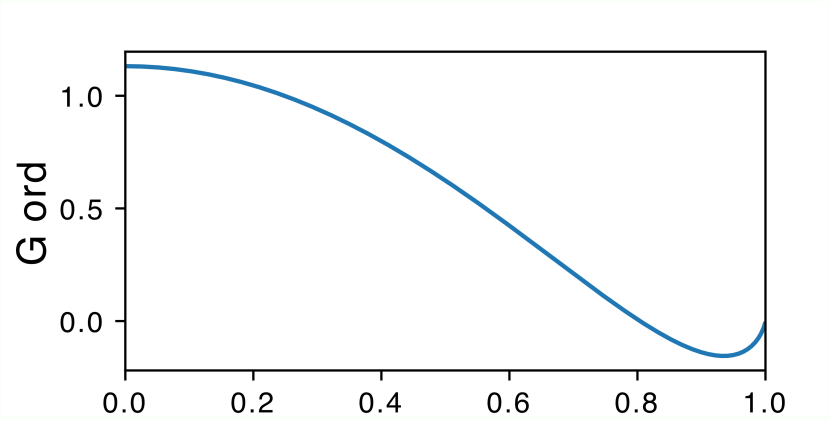

Figure 4.2: ΔG_{ord} vs. Q diagram for sillimanite at 5 kbar - 600 °C.

4.3.3. Finding Q in the Bragg-Williams Model

To determine Q, we can use the Newton-Raphson method to find the minimum of the function described by equation (4.62). The first derivative from the function with respect to Q must be zero, and the second derivative must have a positive value. The search for the parameter involves iterations to adjust Q until the minimum value of ΔG_{ord} is found:

$$Q_{n+1} = Q_n - \frac{f'(x_n)}{f''(x_n)} \tag{4.71}$$

To find the first and second derivatives of ΔG_{ord} with respect to Q, see Worked example 4.4:

$$\begin{aligned}\frac{dG_{ord}}{dQ} = &-H + P \quad (-V + Wv) - 2Q\,(PWv + Wh) + Wh \\ &+ \frac{RTfac}{n+1} \quad \left(-n\log\left(\frac{-Q+1}{n+1}\right) + n\log\left(\frac{Q+n}{n+1}\right)\right. \\ &\qquad \left. +n\log\left(\frac{Qn+1}{n+1}\right) - n\log\left(\frac{n\,(-Q+1)}{n+1}\right)\right)\end{aligned}$$

$$\frac{d^2G_{ord}}{dQ^2} = -2PWv - 2Wh + \frac{RTfac}{n+1}\left(\frac{n^2}{Qn+1} + \frac{n}{Q+n} + \frac{2n}{-Q+1}\right)$$

Worked example 4.4

Derive the expressions for the first and second derivative of G_{ord} with respect to Q.

Using symbolic calculations in *Python* (module *SymPy*):

```
import sympy as sym
Xas, Xasp, Xsb, Xsbp = sym.symbols('Xas, Xasp, Xsb, Xsbp')
n, Q, H, P, V, Wv, Wh, fac, R, T = sym.symbols('n, Q, H, P, V,
                                                Wv, Wh, fac, R, T')
Xas = (1+n*Q)/(n+1)
Xasp = (1-Q)/(n+1)
Xbs = (n-n*Q)/(n+1)
Xbsp = (n+Q)/(n+1)
DHord = H + P*V - Q**2 * (P*Wv+Wh) + Q * (-H+P*(-V+Wv)+Wh)
DSord = -fac * R * (Xas*sym.log(Xas) + Xbs*sym.log(Xbs) + \
         Xasp*sym.log(Xasp**n) + Xbsp*sym.log(Xbsp**n))
DGord =  DHord - T * DSord
dDGord_dQ = sym.simplify(sym.diff(DGord, Q))
d2DGord_dQ2 = sym.simplify(sym.diff(dDGord_dQ, Q))
```

Worked example 4.5

Calculate the ordering parameter Q that minimizes ΔG_{ord} for sillimanite at 5 kbar and 1000 °C. Use the derivatives found in worked example 4.4 to write a function to find the minimum of ΔG_{ord} vs Q.

```
def findQ_NR(P,T,n,H,V,Wh,Wv,fac):
    Q = 0.9999999; R = 0.0083145 #8.31446261815324
    dGdQ = 1
    iterations = 0
    while abs(dGdQ) > 0.0000001 and iterations < 20:
        dGdQ = -H + P * (-V+Wv) - 2*Q*(P*Wv+Wh) + Wh + \
               fac*R*T/(n+1)*(-n*log((-Q+1)/(n+1)) + \
               n*log((Q+n)/(n+1)) PYGo+ n*log((Q*n+1)/(n+1)) - \
               n*log(n*(-Q+1)/(n+1)))
        d2GdQ2 = -2*P*Wv + fac*R*T/(n+1) * \
                  (n**2/(Q*n+1)+\n/(Q+n)+2*n/(-Q+1)) - \
                  2*Wh
        Q = Q - dGdQ/d2GdQ2
        iterations += 1
    print("iterations = ", iterations)
    return Q
```

Worked example 4.5 (cont.)

The function finds $Q = 0.935$ after 12 iterations:

```
h = sil_BW['h']; v = sil_BW['v']; wh = sil_BW['wh']
wv = sil_BW['wv']; fac = sil_BW['fac']; n = sil_BW['n']
Q = findQ_NR(5,1000+273.15,n,h,v,wh,wv,fac)
```

Note the alternative for finding the minimum of the function using the *SciPy* module

```
from scipy import optimize
x0 = [0.9999]
optimize.minimize(Gord, x0, args=(5,1000,n,h,v,wh,wv,fac))
```

For sillimanite, the order parameter is defined as $Q = X_{Al}^{T1} - X_{Al}^{T2}$ (Holland & Powell, 1996); the ordering occurs between two tetrahedral sites. At low temperatures, there is complete ordering ($Q = 1$) where $T1$ is completely occupied by Al while $T2$ is completely occupied by Si. Sillimanite undergoes convergent disordering with temperature increase and there is a rapid decrease in Q at a critical temperature (Holland & Powell, 1996). At the critical temperature (and above), there is complete disorder and both sites are equally occupied by Al and Si ($Q = 0$).

```
P = 5.0; R = 0.0083145
Tc = 2*(1+n)*(wh + wv*P)/R    # 2309.22 K
```

In the example here (1000 °C), Al site proportions are:

```
Xal_t1 = (1.0+n*Q)/(n+1.0)    # 0.9674306690513517
Xal_t2 = (1.0-Q)/(n+1.0)      # 0.032569330948648334
```

4.4. Gibbs Free Energy Calculator for Solid Phases

At the beginning of the chapter, we worked with simplified equations for the Gibbs free energy of pure phases in order to calculate the position of univariant reaction curves. The equations used didn't take into account the pressure dependence of volume (*EOS*) and the ordering behavior of some pure phases. With the last sections in this chapter introducing the necessary concepts to include the equations of state and the ordering behavior of phases, it is now time to work on the construction of a function that calculates the Gibbs free energy at different pressure and temperature conditions.

The following *Python* function is a Gibbs calculator for solid endmembers. The relevant equations were explained in the previous sections.

```
from math import exp, log
def EM_Gibbs(P, T, emData):
    To = 298.15; Po = 0.001; T = T + 273.15; R = 0.0083144626
    Ho = emData.H; So = emData.S; Vo = emData.V
    a = emData.a; b = emData.b; c = emData.c; d = emData.d
    theta = emData.theta
    alpha = emData.alpha; kappa = emData.kappa
    kappa_p = emData.kappa_p; kappa_pp = emData.kappa_pp
    u_o = theta/To
    u = theta/T
    xi_o = (u_o**2)*exp(u_o)/((exp(u_o)-1)**2)
    Pth = alpha*kappa*theta/xi_o*((1/(exp(u)-1))-(1/(exp(u_o)-1)))
    a_eos = (1+kappa_p)/(1+kappa_p+kappa*kappa_pp)
    b_eos = kappa_p/kappa - kappa_pp/(1+kappa_p)
    c_eos = (1+kappa_p+kappa*kappa_pp)/(kappa_p**2+\
                                        kappa_p-kappa*kappa_pp)
    Gvp = P*Vo*(1-a_eos*(1+((1+b_eos*(P-Pth))**(1-c_eos))/\
                         (b_eos*P*(c_eos-1))))
    Gvpo = Po*Vo*(1-a_eos*(1+((1+b_eos*(Po-Pth))**(1-c_eos))/\
                           (b_eos*Po*(c_eos-1))))
    Gv = Gvp - Gvpo

    Int_Cp = a*T + b/2*T**2 - c/T + 2*d*T**(0.5) \
             - (a*To + b/2*To**2 - c/To + 2*d*To**(0.5))
    Int_CpT = (a*log(T) + b*T - c/(2*T**2) - 2*d/(T**0.5) \
              - (a*log(To) + b*To - c/(2*To**2) - 2*d/(To**0.5)))
    Gord = 0
    if emData.flag == 1: #"Landau"
        landau = emData.ordering
        Smax = landau['s_max']; Vmax = landau['v_max']
        Tc_o = landau['tc']
        Tc = Tc_o + (Vmax / Smax) * P
        Qo = ((Tc_o-To)/Tc_o)**(1/4)
        Q4 = ((Tc-T)/Tc_o)
        Q = 0
        if Q4 > 0:
            Q = Q4**(1/4)
        Gord = Smax*Tc_o*(Qo**2 - 1/3*Qo**6) - \
               Smax*(Tc*Q**2 - 1/3*Tc_o*Q**6) - \
               T*Smax*(Qo**2 - Q**2) + P*Vmax*Qo**2
    elif emData.flag == 2: # "BW"
        BW = emData.ordering
        H_BW = BW['h']; V_BW = BW['v']; Wh_BW = BW['wh']
        Wv_BW = BW['wv']; n = BW['n']; fac = BW['fac']
```

(continues on next page)

(continued from previous page)

```
        # Note that this uses the function defined above
        Q = findQ_NR(P,T,n,H_BW,V_BW,Wh_BW,Wv_BW,fac)
        W_BW = Wh_BW + P * Wv_BW
        Tc = 2*W_BW / (R*(1+n))
        Xa_s = (1+n*Q)/(n+1); Xa_sp = (1-Q)/(n+1)
        Xb_s = (n-n*Q)/(n+1); Xb_sp = (n+Q)/(n+1)
        Sord = fac*(-R)* (Xa_s * log(Xa_s) + Xb_s* log(Xb_s) \
               + n*Xa_sp * log(Xa_sp) + n*Xb_sp* log(Xb_sp))
        Hord = H_BW + P*V_BW + Q * ((Wh_BW-H_BW) \
               + P*(Wv_BW-V_BW)) - Q**2 * (Wh_BW+P*Wv_BW)
        Gord = Hord - T * Sord
    G = Ho - T*So + Int_Cp - T*Int_CpT + Gv + Gord
    return G
```

Worked example 4.6

Calculate the change of Gibbs free energy between reactants and products using the full equation at 5 kbar and 500 °C for:

$$Ky = And$$

This problem allows us to compare results obtained from the simplified equation (from the last chapter) with those from the full equation. Here, we simply use our calculator to get $\triangle G_R$:

```
ΔGr = EM_Gibbs(5, 500, dataset["And"]) - \
      EM_Gibbs(5, 500, dataset["Ky"]) # 0.8424
```

Compare this value (0.842 kJ/mol) with the value obtained using the simplified equation (0.882 kJ/mol).

4.5. Univariant Curves

The Gibbs free energy change of a reaction involving pure phases is given by:

$$\triangle_r G_{P,T} = \triangle_r H^0_{T_0} + \int_{T_0}^{T} \triangle_r C_P dT - T \triangle_r S^0_{T_0} - T \int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT + \int_{P_0}^{P} \triangle_r V \, dP \tag{4.72}$$

For simplicity, the entropy contribution from pure phase ordering is ignored. Under equilibrium conditions, $\triangle_r G_{P,T} = 0$, and the resulting equation can be solved for pressure or temperature. Rearranging the equation is not an option since the equation is a complex function of P and T; instead a numerical method must be used. A reasonable method is the application of the Newton-Raphson algorithm. There, we need the derivative of the function either with respect to T or to P. Alternatively, we could use *SciPy* functions to find the zero root of the equation.

Here, we will follow the first approach. In the function above, the definite integrals have a constant lower limit (T_0 and P_0); the derivative of the integral is then the terms within the integral, evaluated at the upper bound (the variable T or P), for example:

$$\frac{\partial \int_{T_0}^{T} \triangle_r C_P dT}{\partial T} = (\triangle_r C_P)_T \tag{4.73}$$

- For fixed temperatures, the Newton-Raphson algorithm needs $\partial G/\partial P$:

$$\frac{\partial \triangle_r G_{P,T}}{\partial P} = \triangle_r V_{P,T} \tag{4.74}$$

Note that $V_{P,T}$ for solid phases is given in equation (4.29)

- For fixed pressures, the Newton-Raphson algorithm needs $\partial G/\partial T$:

$$\frac{\partial \triangle_r G_{P,T}}{\partial T} = (\triangle_r C_P)_T - \triangle_r S^0_{T_0} - \int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT - (\triangle_r C_P)_T \tag{4.75}$$

$$\frac{\partial \triangle_r G_{P,T}}{\partial T} = -\triangle_r S^0_{T_0} - \int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT \tag{4.76}$$

The *Python* function to calculate the Gibbs free energy needs to be modified to include calculations for $\partial G/\partial P$ and $\partial G/\partial T$ and return these two values.

```
    Vt = Vo * (1-a_eos*(1-((1+b_eos*(P-Pth))**(-c_eos))))
    dG_dT = -So - Int_CpT
    G = Ho - T*So + Int_Cp - T*Int_CpT + Gv + Gord
    return (Vt, dG_dT, G)
```

Rename this function:

```
def EM_Gibbs_dif(P, T, emData):
    ...
    return (Vt, dG_dT, G)
```

And write a convenience function that will return only the Gibbs free energy; this is done for compatibility with previous written code:

```
def EM_Gibbs(P, T, emData):
    (_, _, G) = EM_Gibbs_dif(P, T, emData)
    return G
```

Worked example 4.7

Write a *Python* function to calculate the position of the *GASP* reaction ($3\ An = Gr + 2\ Ky + Qz$) in a pressure vs. temperature diagram. Plot the reaction curve between 450°C and 750°C.

---

1. Write the *Python* function. This function calls the **EM_Gibbs_dif()** function to get the Gibbs free energy and its derivatives for each phase in the reaction. Then, it calculates a residual of $\triangle_r G_{P,T}$ and a ponderator to adjust the pressure. The loop stops when the calculated residual gets below the tolerance value or when the maximum number of iterations is reached.

```
def gasp(T): # 3 an = gr + 2 ky + q
    tol = 1e-4
    maxIter = 10
    P = np.zeros(len(T))
    for i in range(len(T)):
        Ti = T[i]
        Pi = 10
        iter = 0
        res = 100
        while abs(res) > tol and iter < maxIter:
            iter += 1
            # calculate Gibbs and derivatives
            (Vt_an, dG_dT_an, G_an) = EM_Gibbs(Pi,Ti,
                                               dataset["An"])
            (Vt_gr, dG_dT_gr, G_gr) = EM_Gibbs(Pi,Ti,
                                               dataset["Gr"])
            (Vt_ky, dG_dT_ky, G_ky) = EM_Gibbs(Pi,Ti,
                                               dataset["Ky"])
            (Vt_q, dG_dT_q, G_q) = EM_Gibbs(Pi,Ti,
                                            dataset["Q"])
            # calculate residual and ponderator
            res = G_gr + 2 * G_ky + G_q - 3 * G_an
            ponder = Vt_gr + 2 * Vt_ky + Vt_q - 3 * Vt_an
            # adjust P
            Pi -= res/ponder
        P[i] = Pi
    return P
```

Worked example 4.7 (cont.)

2. The following script plots the equilibrium pressures at specified temperatures in the range 450 to 750 °C.

```
T = np.arange(450,755,5)
fig, ax = plt.subplots()
ax.plot(T, gasp(T))
ax.set_ylabel('Pressure (kbar)', fontsize=14)
ax.set_xlabel('Temperature ($^{\circ}$C)', fontsize=14)
```

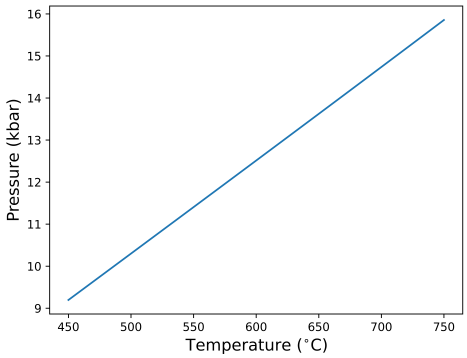

Figure 4.3: Reaction curve for the *GASP* reaction (3 $An = Gr$+2 Ky+Qz) calculated using a Newton-Raphson algorithm.

4.6. Gibbs Energy Minimization for Systems with Pure Phases

The problem is stated as follows: given the Gibbs free energy of all possible minerals and the chemical composition of a system (rock), determine the abundances of minerals that minimize the Gibbs free energy of the system. To solve the problem, we need the mass conservation equations for each component (oxide or elemental proportions). An additional limitation is that the abundances of phases cannot be negative.

Schematically, Figure 4.4 shows an example with three phases in a system with two components. At composition x_1, phases A and B are stable, whereas at composition x_2, B and C are stable. The phase proportions can be derived using the lever rule, i.e., the line joining G for stable phases is

split in two segments by the vertical line representing the compositions, and the relative length of the segments indicates the abundances of the phases.

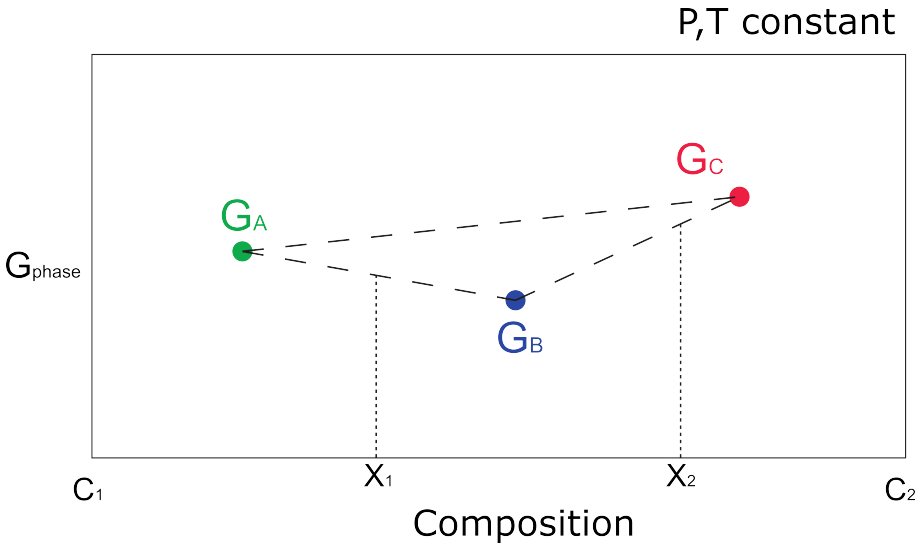

Figure 4.4: Schematic diagram (at constant pressure and temperature) showing the stability of three pure phases in a two-component system. Broken lines represent the Gibbs free energy of a system with the two phases joined by the line. The broken line joining A and C is always above the other two broken lines, i.e., it always has a higher Gibbs free energy; this association is not stable at the specified pressure and temperature.

The mathematical treatment followed here is presented in Albarède (1996); the function that must be minimized is:

$$G = \sum_{j=1}^{p} n_j g_j = \mathbf{n}^{\mathrm{T}}\mathbf{g} \tag{4.77}$$

with:

- **n**: vector with the abundance of stable phases; n_j is the number of moles of each mineral.
- **g**: vector with the Gibbs free energy of formation at P and T of interest ($\Delta G^0_{P,T}$) for each mineral (g_j).

The conservation equations dictate that the dot product of the compositional matrix for stable phases with the vector having the number of moles of stable phases must be equal to the composition of the system:

$$\mathbf{A}^{\mathrm{T}}\mathbf{n} = \mathbf{q} \tag{4.78}$$

where $\mathbf{q}$ is the vector with the composition of the rock (this vector has a dimension s and it contains the q_i mole fractions of each component), and $\mathbf{A}$ is a matrix with the composition of the mineral phases (pure) that may be present, the number of possible phases is p, where $p \geq s$. Under equilibrium conditions, the system with s components cannot consist of more than s mineral phases. This consideration allows splitting the vector $\mathbf{n}$ into two parts: (i) the first is a vector with the base variables (phases considered in the calculation of the Gibbs free energy of the system) with dimension s, represented by $\mathbf{n_B}$, and (ii) the second is a vector with the so-called free variables with dimension $p - s$, represented by $\mathbf{n_F}$. The matrix $\mathbf{A}$ is also split into two parts: an upper $s \times s$ compositional matrix for base variables ($\mathbf{A_B}$) and a lower $(p - s) \times s$ compositional matrix for free variables ($\mathbf{A_F}$). For the vector $\mathbf{n}$ to be a possible solution, the following must be true:

$$\mathbf{n}^{\mathrm{T}}\mathbf{A} = \left[\mathbf{n}_{\mathbf{B}}^{\mathrm{T}}, \mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\right] \begin{bmatrix} \mathbf{A_B} \\ \mathbf{A_F} \end{bmatrix} = \mathbf{n}_{\mathbf{B}}^{\mathrm{T}}\mathbf{A}_{\mathbf{B}}^{\mathrm{T}} + \mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\mathbf{A_F} = \mathbf{q}^{\mathrm{T}} \tag{4.79}$$

From the equation above, we get:

$$\mathbf{n}_{\mathbf{B}}^{\mathrm{T}} = \mathbf{q}^{\mathrm{T}}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}} - \mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}} \tag{4.80}$$

Once this relationship is established, the algorithm must seek to change $\mathbf{n_B}$ and $\mathbf{n_F}$ in such a way that G decreases. This is achieved by changing a base variable for a free variable. To find which variables must be changed, we need to find the differential of the last equation:

$$\partial\mathbf{n}_{\mathbf{B}}^{\mathrm{T}} = -\partial(\mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}}) \tag{4.81}$$

and the differential of equation for G:

$$\partial\mathbf{G} = \partial\mathbf{n}^{\mathrm{T}}\mathbf{g} = \left[\partial\mathbf{n}_{\mathbf{B}}^{\mathrm{T}}, \partial\mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\right] \begin{bmatrix} \mathbf{g_B} \\ \mathbf{g_F} \end{bmatrix} = \partial\mathbf{n}_{\mathbf{B}}^{\mathrm{T}}\mathbf{g_B} + \partial\mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\mathbf{g_F} = \partial\mathbf{n}_{\mathbf{F}}^{\mathrm{T}}\left(\mathbf{g_F} - \mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}}\mathbf{g_B}\right) \tag{4.82}$$

The term in parentheses, $\mathbf{g_F} - \mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}}\mathbf{g_B}$, is the gradient of G with respect to the free variables.

Changes are made so that the conservation equations are satisfied. Note that each component of $\mathbf{n_F}$ is zero and can only be incremented. $\partial\mathbf{G}$ is only negative if the gradient $\mathbf{g_F} - \mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}}\mathbf{g_B}$ has at least one negative component. G is lowered by increasing the number of moles of the i component of $\mathbf{n_F}$ associated with the most negative value of $\mathbf{g_F} - \mathbf{A_F}\mathbf{A}_{\mathbf{B}}^{-\mathbf{1}}\mathbf{g_B}$. The i component

then moves from free to base variable, and this will cause that one component in the base variables will reach a mode of zero. The first component that reaches a zero modal proportion is given by the minimum of:

$$\frac{\mathbf{q}^{\mathrm{T}}\mathbf{A}_{\mathbf{B}}^{-1}}{[\mathbf{A}_{\mathbf{F}}\mathbf{A}_{\mathbf{B}}^{-1}]_i} = \frac{\mathbf{n_B}}{\mathbf{u_i}} \tag{4.83}$$

where $\mathbf{u_i}$ is the ith row of $\mathbf{A_F}\mathbf{A_B^{-1}}$. Note that during the selection of the initial assemblage we must ensure that all elements of $\mathbf{n_B}$ be positive.

Worked example 4.8

Find the equilibrium mineral association at 10 kbar and 700 °C for a *KAS* system with the following composition: $n_{SiO_2} = 0.8$, $n_{K_2O} = 0.05$, $n_{Al_2O_3} = 0.15$. Consider the phases kyanite, corundum, quartz, kalsilite, leucite, and microcline (Figure 4.5).

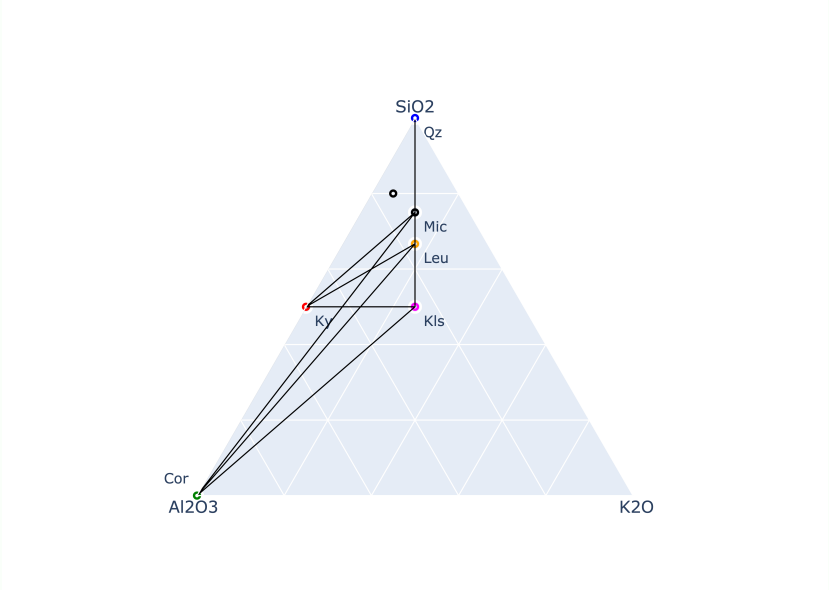

Figure 4.5: Triangular compatibility diagram for the KAS system. The considered mineral phases are joined by compatibility lines. The system composition is represented by an open circle in the upper part of the triangle.

Worked example 4.8 (cont.)

1. Create matrices and vectors used in the minimization equations:

```
# compositional matrix
#              SiO2 Al2O3 K2O
A=np.matrix([[1,   1,   0], # Ky
             [0,   1,   0], # Cor
             [1,   0,   0], # Qz
             [1, 0.5, 0.5], # Kls
             [2, 0.5, 0.5], # Leu
             [3, 0.5, 0.5]])# Mic
# Gibbs free energy
g = np.array([EM_Gibbs(10,700,dataset["Ky"]), \
              EM_Gibbs(10,700,dataset["Cor"]), \
              EM_Gibbs(10,700,dataset["Q"]), \
              EM_Gibbs(10,700,dataset["Kls"]), \
              EM_Gibbs(10,700,dataset["Lc"]), \
              EM_Gibbs(10,700,dataset["Mic"])])
# rock compositional vector q
q=np.array([0.8,0.15,0.05])
```

2. Select an starting basis (*Ky*+*Qz*+*Kls*) and calculate the gradient of G relative to free variables ($\mathbf{g_F} - \mathbf{A_F A_B^{-1} g_B}$):

```
# Starting basis Ky, Qz, Kls
b_phases = np.array([0, 2, 3])
f_phases = np.delete(np.arange(6), b_phases)
A_b = A[b_phases]
g_b = g[b_phases]
A_f = A[f_phases]
g_f = g[f_phases]
# calculate A_f * (A_b)-1
a_fb = np.dot(A_f, np.linalg.inv(A_b))
# the constrained gradient of G relative to the free variables
G_grad = g_f.T - np.dot(a_fb, g_b)  # 2.1 -8.3 -21.9
```

3. The most negative component (-21.9) is the component $A[5]$ (*Mic*). This means that microcline must move from free to base variable. The number of moles of the minerals is divided by the row of $\mathbf{A_F A_B^{-1}}$ corresponding to the most negative value of the gradient ($\mathbf{n_B/u_i}$) to determine which will be the first to reach a value of zero when *Mic* enters the assemblage and its proportion increases:

Worked example 4.8 (cont.)

```
i_min = np.argmin(G_grad) # index of most neg. comp.
u_i = b_fb[i_min]
n_b = np.dot(q.T, np.linalg.inv(A_b))
j_out = np.argmin(n_b/u_i) # index of comp. with mode -> 0
```

Component $A[3]$ (*Kls*) will be the first to reach zero. Therefore, we exchange *Kls* with *Mic* and repeat the steps above until all elements of the gradient in (2) are positive. For the next step we can write:

```
temp = b_phases[j_out]
b_phases[j_out] = f_phases[i_min]
f_phases[i_min] = temp
```

The second iteration produces only positive values for the gradient of G. The Gibbs free energy of the system is −1071.84 kJ with mole proportions $Ky = 0.17$, $Qz = 0.67$, and $Mic = 0.17$.

Note: watch out for the number of moles of the phases and for singular matrices. For example, try the starting assemblage $Ky + Qz + Leu$.

Chapter 5

Phase Equilibria in Systems with Solid Solutions

$$Y = \sum_i n_i y_i$$

5.1. Partial Molar Properties, Chemical Potential, and Darken's Equation

The properties of dissolved substances, which are part of a solid solution, are known as apparent or partial molar properties. If Y is an extensive thermodynamic property (variable) of a system, then the partial molar property y_i represents the rate of change of Y with respect to the number of moles of component i, that is:

$$y_i = \left(\frac{\partial Y}{\partial n_i}\right)_{P,T,n_{j(i\neq j)}} \tag{5.1}$$

Thus, the partial molar Gibbs free energy is:

$$g_i = \left(\frac{\partial G}{\partial n_i}\right)_{P,T,n_{j(i\neq j)}} = \mu_i \tag{5.2}$$

which represents the change in Gibbs free energy with respect to the number of moles of a component, while holding pressure, temperature, and other components constant. This function is known as the chemical potential of the component i.

For a solid solution, the relationship between an extensive property and the molar partial properties of the components at constant pressure and temperature is given by:

$$Y = \sum_i n_i y_i \tag{5.3}$$

Applying this equation, we see that, in more general terms, the Gibbs free energy of a system at constant P and T, is the sum of the free energies of all the phases present, defined by the chemical potential of its components:

$$G = \sum_i n_i \mu_i \tag{5.4}$$

However, what is usually known are the molar properties (Y_m) as a function of the mole fractions or concentrations of the components (X_i). Darken (1950) presented an equation relating the partial molar quantity of a component (y_i) and the corresponding molar property (Y_m) of a multicomponent solution, derived from the generalized Gibbs-Duhem equation (see below):

$$y_i = Y_m + (1 - X_i)\left(\frac{\partial Y_m}{\partial X_i}\right)_{P,T} \tag{5.5}$$

This equation applies to a multicomponent solution since $\partial Y_m/\partial X_i$ is a partial derivative, meaning that all ratios of mole fractions, except those involving X_i, are held constant. In other words, solid solutions are reduced to pseudobinaries by taking the partial derivative with respect to one component and keeping the relative proportions of the other components constant. The geometric interpretation of this equation is that the partial molar properties of the two components at a given composition are given by the intercepts of the tangent to the curve at both extremes of the pseudobinary. In Figure 5.1, the intercepts of a tangent to the curve at point $X_A = 0.3$ are taken at $X_A = 0$ and $X_A = 1$.

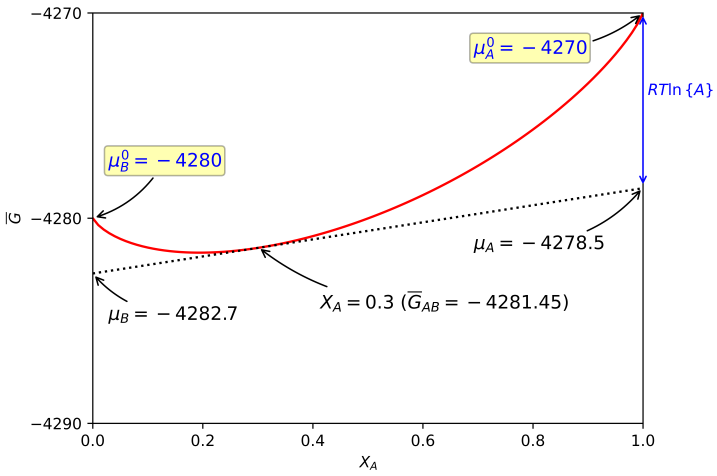

Figure 5.1: Gibbs free energy for a phase as a function of chemical potential of its components.

5.2. The Chemical Potential and the Gibbs Free Energy Equation in Multicomponent Phases (Solid Solutions)

In closed systems, the derivative of the Gibbs free energy is a function of pressure and temperature only. In open systems, the number of moles (n) of the involved substances must also be considered, $dG = f(T,\ P,\ n)$. If the system is reversible, open, and simple, the differential form of the Gibbs

free energy equation becomes:

$$dG = -SdT + VdP + \sum \left(\frac{\partial G}{\partial n_i}\right)_{P,T} dn_i \tag{5.6}$$

As seen earlier, the function $\partial G/\partial n_i$ is a partial molar property known as the chemical potential. This function is represented by the tangent to the curve at any point on a Gibbs free energy vs. composition graph (Figure 5.1). That is, the Gibbs free energy of a phase can be expressed as a function of the chemical potential of its components through the Darken equation.

5.3. The Gibbs-Duhem Equation

The Gibbs-Duhem equation relates the intensive variables of a system in equilibrium:

$$VdP - SdT - \sum n_i d\mu_i = 0 \tag{5.7}$$

For a two-component system at constant P and T: $n_1\mu_1 + n_2\mu_2 = 0$. It can be seen here that if μ_1 is increased, then μ_2 must decrease, and therefore we have that in this two-component system, only one is independent. More generally, in a n-component system, $n - 1$ are independent. The common form of the Gibbs-Duhem equation is obtained by dividing with the total number of components N, at P and T constant:

$$\sum_i X_i \mu_i = 0 \tag{5.8}$$

5.4. Raoult's Law and Henry's Law

These two laws describe the ideal behavior of solvents and solutes in dilute mixtures. *Raoult's Law* describes the behavior of a solvent phase in a dilute solution (Figure 5.2). This law establishes that the vapor pressure of a component in a mixture depends on its molar fraction in the mixture (X_i), mathematically:

$$P_i = X_i P_i^0 \tag{5.9}$$

where P_i is the vapor pressure of component i in the mixture, and P_i^0 is the vapor pressure of component i in its pure state.

Henry's Law describes the behavior of a solute in an ideal solution (Figure 5.2):

$$P_j = k_H X_j \tag{5.10}$$

where P_j is the vapor pressure of the trace component j, and k_H is the Henry's law constant.

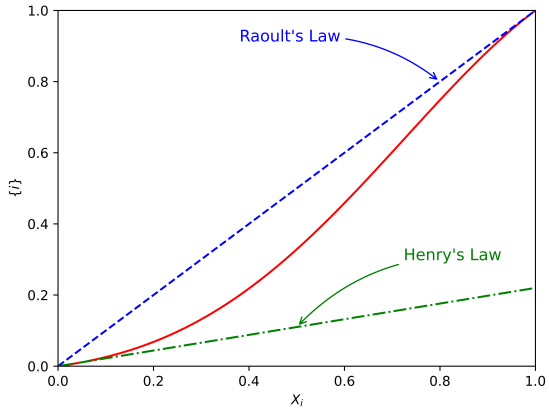

Figure 5.2: Activity of a component in a non-ideal binary mixture as a function of composition; the curve represents mixing occurring in only one site with multiplicity equals to 1. Henry's law of ideal mixing applies at low proportion of component a (acting as the solute), and Raoult's law of ideal mixing applies at high proportions of a (acting as the solvent phase). The Henry's law constant is the slope of the green line (here, it is 0.22).

Worked example 5.1

Estimate graphically the regions of a binary solid solution where the ideal laws apply for a component a. Use the equation for ideal activity $\{a\}_{ideal} = X_a$. We have not yet introduce the concepts of activity and non-ideality in solid solutions. For the purpose of the exercise here, use $Wh = -40$ and $T = 600°C$ in the following equation to determine the departure from ideal behavior of a solid solution:

$$activity = e^{(-1/(R*T)*((1-Xa)*(-Xb)*Wh))} * X_a$$

1. First, write a *Python* function to calculate the real activity. This function takes as parameters: component (endmember) proportions, W_h and temperature.
2. Plot activity versus composition and zoom in at the extremes to estimate graphically where the *Henry's* and *Raoult's* laws apply.

Worked example 5.1 (cont.)

```
# (1):
import numpy as np
R = 0.0083144
def real_activity(x, Wh, T):
    T = T + 273.15
    Xa = x
    Xb = 1 - x
    activity_a = np.exp(-1/(R*T)*((1-Xa)*(-Xb)*Wh)) * Xa
    return activity_a
# (2):
import matplotlib.pyplot as plt
steps = 100
x = np.arange(0.00000000001,1.0000000001,1/steps)
fig, (ax1, ax2) = plt.subplots(2,1)
ax1.plot(x, real_activity(x, -40, 600), 'r-')
ax1.plot([0,1], [0,1], 'b--')
ax1.plot([0,1], [0,0.005], 'g-.')
ax1.set_ylabel(r'$\{a\}$')
ax1.set_xlabel(r'$X_i$')
ax1.set(ylim=(0.9,1), xlim=(0.9,1))
ax2.plot(x, real_activity(x, -40, 600), 'r-')
ax2.plot([0,1], [0,1], 'b--')
ax2.plot([0,1], [0,0.006], 'g-.')
ax2.set_ylabel(r'$\{a\}$')
ax2.set_xlabel(r'$X_i$')
ax2.set(ylim=(0.9,1), xlim=(0.9,1))
```

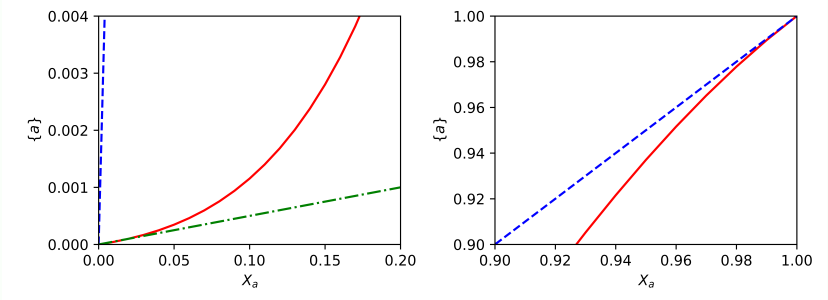

Figure 5.3: Zoom of an activity vs. composition plot for a non-ideal mixture to estimate the regions where *Raoult's and Henry's laws* apply. The green curve uses equation (5.10) with the parameter k_H adjusted by trial and error. Graphically, *Henry's law* apply in the range $X_a 0.03$ and *Raoult's law* apply in the range $X_a > 0.99$.

5.5. Fugacity and Activity

The differential form of the Gibbs free energy equation for a simple, closed, and reversible system at constant temperature becomes:

$$dG = V\,dP \tag{5.11}$$

According to the ideal gas law, for 1 mole of gas:

$$V = \frac{RT}{P} \tag{5.12}$$

Since we have the Gibbs free energy of one mole (molar Gibbs free energy), we can express the chemical potential by combining the last two equations:

$$d\mu = RT\left(\frac{dP}{P}\right) \tag{5.13}$$

$$\int_{P_0}^{P} d\mu = \int_{P_0}^{P} RT\left(\frac{dP}{P}\right) \tag{5.14}$$

$$\mu - \mu_0 = RT(\ln P - \ln P_0) \tag{5.15}$$

$$\mu - \mu_0 = RT\ln\left(\frac{P}{P_0}\right) \tag{5.16}$$

This equation can be generalized for non-ideal gases using fugacities (f) instead of partial pressures. The fugacity is the vapor pressure of a gas (i.e., the tendency of a substance to escape):

$$f_i = P_i\Gamma_i \tag{5.17}$$

where Γ_i is the fugacity coefficient, which corrects the partial pressure for non-ideal behavior (in an ideal gas, high T or low P, $\Gamma_i = 1$), so for non-ideal gases:

$$\mu = \mu_0 + RT\ln\left(\frac{f}{f_0}\right) \tag{5.18}$$

The relationship between fugacity and standard state fugacity is defined as the activity of a component i:

$$\{i\} = \frac{f}{f_0} \tag{5.19}$$

then

$$\mu = \mu_0 + RT \ln\{i\} \tag{5.20}$$

The concept of activities in gaseous mixtures can be extended to aqueous solutions, where the activities are expressed in terms of concentrations (commonly in molality, m_i) and an activity coefficient (γ_i):

$$\{i\} = \gamma_i m_i \tag{5.21}$$

and to solid solutions, where the activities are expressed in terms of thermodynamic mole fractions or ideal activities ($\{i\}_{ideal}$) and a rational activity coefficient (λ_i):

$$a = \lambda_i \{i\}_{ideal} \tag{5.22}$$

As seen from equation (5.20), the difference between the chemical potential of a component in a phase (μ_i^{phase}) and the chemical potential of the component in its standard state (μ_0) is explained through the concept of activity (see also Figure 5.1).

5.6. Standard States

According to the generalized chemical potential equation (5.20), we can write for a component i in a phase A:

$$\mu_i^A = \mu_{i,0}^A + RT \ln\{i\}^A \tag{5.23}$$

From this equation, and knowing that the chemical potential is the partial molar Gibbs free energy, we can see that the chemical potential of a component i in a phase A is a function of temperature, pressure, and composition. In equation (5.23), the first term is independent of the composition of phase A, while the second term depends on the composition. The first term is known as the *standard chemical potential*; the standard state to which it refers is a selected state of temperature, pressure, and composition for the phase of interest.

The formal thermodynamic restriction of the standard state generally includes the temperature of interest and a fixed composition, but there is freedom in the selection of the fixed composition and the pressure. This is an arbitrary step that divides the chemical potential into a composition-dependent term ($RT \ln\{i\}$) and a composition-independent term ($\mu_{i,0}$).

The real activity is calculated through an ideal activity factor and a activity coefficient factor, thus:

$$\mu_i = \mu_{i,0} + RT \ln \left(\{i\}_{ideal} \lambda_i \right) \tag{5.24}$$

Where $\lambda_i \rightarrow 1$ when $X_i \rightarrow 1$ and $\lambda_i \rightarrow constant$ when $X_i \rightarrow 0$. It is important to have in mind two points: (i) regardless of the chosen standard state, the activity of a component in its standard state is always unity (note that this is no the same as saying that the activity of a component in its pure state is unity, because this depends on the choice of standard state); and (ii) the final result of a thermodynamic calculation must be independent of the choice of standard state. Following the rules for establishing a standard state, there are a variety of options, and the final selection is dictated by trying to simplify the calculations. As we will see below, *Henry's* and *Raoult's* laws play an important role in the selection of appropriate standard states to simplify calculations.

A common approach is to select a standard state where the activity of a component is equal to one and the natural logarithm of the activity is equal to zero. The second term of equation (5.23) takes into account the difference between the chemical potential of the component in a phase and its *standard chemical potential* (see Figure 5.1).

A common choice for the standard state is that of a pure phase (real or hypothetical) at the conditions of P and T of interest. In a binary mixture, the behavior of the chemical potential for a component in this standard state can be visualized as having three regions (Figures 5.2 and 5.4). In the first region, where the component proportion is high, close to the pure endmember ($X_i^A \rightarrow 1$), there is a linear relationship between activity and composition, such that $\{i\}^A = X_i^A$. This is the *Raoult's* law region. At the other end of the spectrum, where the component is highly diluted ($X_i \rightarrow 0$), there is also a linear relation between activity and composition, but with a proportionality constant (*Henry's law* constant, k_H), this is the region of *Henry's law* where $\{i\} \rightarrow k_H * X_i$. In both cases, the limiting lines in a μ_i vs $\log X_i$ plot have slopes equal to $R \cdot T$.

The third region, in the middle of Figure 5.4, is a non-ideal region, where μ_i vs. $\log X_i$ is a curve, described by an equation containing an activity coefficient, which is also a function of the composition but different from 1 (*Raoult's law* region) or k_H (*Henry's law* region):

$$\mu_i^A = \mu_{i,0}^A + RT \ln(\lambda_i^A \{i\}_{ideal}^A) \tag{5.25}$$

Another choice for the standard state applies to situations where there is no pure endmember and actually it does not make sense to have this pure

member as is in the case of aqueous solutions (e.g., a pure electrolyte in the solvent structure, like $CaCl_2$ in H_2O). Although this state is commonly applied to electrolytic solutions because the dissociation of the electrolytes does not allow to have the pure electrolyte in water, this standard state can also be applied to solid mixtures. For example, many phases contain Fe^{+3}, but in most cases, a phase composed solely of Fe^{+3} does not exist, making it impossible to determine the standard chemical potential of the pure component. The derivation of this standard state is based on the fact that it is possible to make measurements of the chemical potential when the solution is diluted (in the region of Henry's law). This is a hypothetical state obtained by extrapolation along the "line of Henry's Law" as $X_i \to 1$ (or $m_i \to 1$), in Figure 5.4, this extrapolation gives us a value of G_i^{0*}. In this situation, G_i^0 cannot be determined; we just know that G_i^{0*} is a function of $RT \log k_H$ and since this value depends on composition, the value of G_i^{0*} depends on the endmember in which the component i is diluted. This dependence is the reason why this standard state is not useful for trace element geochemistry.

A third case arises when a pure component in a solid solution A cannot be studied, but it can be studied in another solid solution B. Here, the standard state is defined as the pure endmember in a different structure, at the pressure and temperature of interest. This is used, for example, for silicate liquid endmembers.

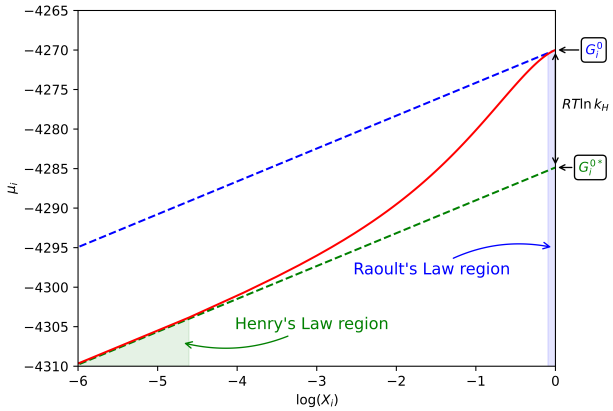

Figure 5.4: Chemical potential vs. natural log of composition to show the three regions of the standard state with a pure phase (real or hypothetical) at the conditions of P and T of interest. Figure was drawn for a hypothetical two endmember solid solution with mixing on one site with multiplicity equals to 1. μ_A = -4270, T = 500 K, W_{AB} = −15.0 Kj, k_H = 0.029.

Two other possible states are the standard state of a condensed pure component at 1 bar and the temperature of interest ($\{i\} \rightarrow X_i$ at 1 bar), and the standard state in which the fugacity of the pure gas phase is one and activity equals to fugacity (usually assuming that at P = 1 bar, fugacity = 1). For the condensed pure component at the 1 bar standard state, the activity at any other pressure is given by:

$$RT \ln\{i\}_0(P,T) = G_{i,0}(P,T) - G_{i,0}(1\,bar,T) = \int_1^P V_{i,0} dP \qquad (5.26)$$

5.7. Ideal Solid Solutions

5.7.1. Molar Properties in Ideal Mixtures

When the species are ideally mixed, the molar enthalpy and the molar volume of the resulting solution are equal to the sum of the contributions of the individual species. The thermodynamic properties of these solutions are then given by

$$\triangle H_{mix} = 0, \;\; \triangle V_{mix} = 0 \qquad (5.27)$$

However, the molar entropy of the solution is greater than the sum of the entropies of the species (the act of mixing changes the ordering state of the system).

5.7.2. The Entropy of Mixing and Activities in Ideal Mixtures

The microscopic interpretation of entropy is given by the *Boltzmann* relation, which states that each macroscopic state of a system is associated with a certain number of microscopic states. The entropy of the macroscopic state is proportional to the natural logarithm of the number of microscopic states. In terms of Plank

$$S_{mix}^{ideal} = S_{config.}^{ideal} = k \ln \Omega \qquad (5.28)$$

where Ω represents the number of possible microscopic crystalline states that occur in a mineral, and k is Boltzmann's constant ($1.381 \times 10^{-23} JK^{-1}$). For an ideal binary solution $A - B$ we have that:

$$\Omega = \frac{N!}{N_A! N_B!} \qquad (5.29)$$

where N is the total number of particles, and N_A and N_B are the numbers of particles of components A and B, respectively.

For example, if there is a *macroscopic* state with nine particles, one of different characteristics than the other eight, there are nine possible *microscopic* states, as shown in Figure 5.5.

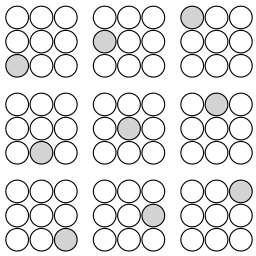

Figure 5.5: Sketch showing the nine possible states for a bidimensional square assemble with eight white particles and one gray particle (represented as circles).

5.7.2.1. Derivation of a General Expression for Activities in Solid Solutions

N_A and N_B can be expressed in terms of mole fractions: $N_A = NX_A$ and $N_B = NX_B$ (where $X_A + X_B = 1$). The configurational entropy of a completely disordered (ideal) crystal of a binary mixture is then:

$$S_{config.}^{ideal} = k \ln \frac{N!}{(NX_A)!(NX_B)!} = k\,[\ln N! - \ln(NX_A)! - \ln(NX_B)!] \quad (5.30)$$

Since N is very large, we can use *Stirlings' theorem* ($lnN! = N\,ln(N) - N$) in equation (5.30):

$$S_{config.}^{ideal} = -Nk\,(X_A \ln X_A + X_B \ln X_B) = -R\,(X_A \ln X_A + X_B \ln X_B) \quad (5.31)$$

Where R is the gas constant. Equation (5.31) is valid only if the multiplicity of the site where mixing occurs is 1 (that is, there is only one atom per crystallographic site). Equation (5.31) can be generalized for mixtures of n components that occur in a crystallographic site of multiplicity q (number of equivalent sites):

$$S_{config.}^{ideal} = -qR \sum_i X_i^j \ln X_i^j \quad (5.32)$$

where X_i^j is the mole fraction of the ion i at the site j and q_j is the multiplicity of the site where mixing occurs. In the unit formula of a phase, this corresponds to the number of cations that can occupy the site. If there is more than one crystallographic site, each of them can be considered independently, this is the ideal mixing on sites model (*IMOS*). The mixing entropy is given by the equation (5.32) generalized for crystal structures with several crystallographic sites:

$$\Delta S_{\text{ideal}} = -R \sum_j q_j \sum X_i^j \ln X_i^j \tag{5.33}$$

The molar Gibbs free energy of a mixture is the sum of the contributions of the Gibbs free energy of each of the mixed species (mechanical mixing) plus a delta caused by the chemical mixture. For ideal mixtures, there is only an entropic contribution to this delta. This relationship is expressed as:

$$\overline{G} = \sum_i X_i \mu_i^0 - \Delta S_{\text{ideal}} * T \tag{5.34}$$

$$\overline{G} = \sum_i X_i \mu_i^0 + RT \sum_j q_j \sum_i X_i^j \ln X_i^j \tag{5.35}$$

With equation (5.20), the molar Gibbs free energy of an ideal mixture can be expressed as:

$$\overline{G} = \sum_i X_i^A \mu_{i,0}^A + RT \sum_i X_i^A \ln\{i\}_{ideal}^A \tag{5.36}$$

Note that X_i^A is the proportion of the components in phase A. This term should not be confused with the term X_i^j which refers to molar fractions of ions in the crystallographic sites of the phase A. $\{i\}_{ideal}$ corresponds to the ideal activity of component i in the mixture. Then, the Gibbs free energy of an ideal mixture is given by the sum of the Gibbs free energy of the mechanical mixture plus the energy produced by the effect of the mixing entropy.

$$G_{ideal} = G_{mec} + G_{idealmix} \tag{5.37}$$

Equations (5.35) and (5.36) can be used to derive the relation (see the worked example below):

$$\{i\}_{ideal}^{mix} = \prod_j \prod_i (X_i^j)^{q_j} \tag{5.38}$$

However, to guarantee that the activity of a component in a pure phase is equal to 1, it is necessary to include a correction factor or normalization constant (C):

$$a_i^{mix} = C \prod_j \prod_i (X_i^j)^{q_j} \tag{5.39}$$

Worked example 5.2

Examine the proportions of components as shown in Figure 5.6 for a solid solution formed by the mixture of cations A and B at crystallographic site *1* and cations C and D at crystallographic site *2*.

According to equation (5.36), the proportion of endmembers in a mixture play an important role in the calculation of the Gibbs free energy of the mixture. For many mixtures, it is relatively easy to derive equations for proportions. For example, in a binary mixtures of albite and anorthite (plagioclase) $X_{ab} = X_{Na}$ and $X_{an} = 1 - X_{ab} = X_{Ca}$. These equations are easily found because all the sodium content in the mixture comes from the albite endmember and with the requirement that the sum of the proportions must be equal to 1. Note that in this case the two proportions vary between 0 and 1, but in general, proportions do not necessarily have to be positive values.

In the solid solution of Figure 5.6, four endmembers can be defined: AD, BD, CA, and CB. However, only three are independent endmembers. The selection is arbitrary, in the example AD, BD, and CA were selected. Notice that in the diagram on the left the proportions of all endmembers are positive, while in the diagram on the right, the proportion of one endmember is negative ($X_{AD} < 0$). In general, endmember proportions are calculated by transforming a set of old components (mole fractions at crystallographic sites) to a set of new components (proportions) through the use of linear algebra as seen in Chapter 3.

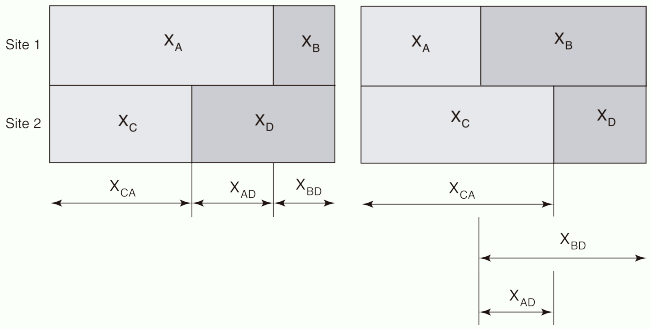

Figure 5.6: Sketch with proportions of endmembers (e.g., X_{CA}) based on cation distribution in two crystallographic sites (e.g., X_C and X_A)

Worked example 5.3

Derive the activity expressions for the plagioclase *2T* model.

The cation distribution in this model is as follows:

$$(Ca, Na)^{M}(Al, Si)_2^{T2}Si_2^{T1}O_8$$

$$Ab : (Na)^{M}(Al_1Si_1)^{T2}Si_2^{T1}O_8,\ An : (Ca)^{M}(Al_2)^{T2}Si_2^{T1}O_8$$

In order to find the activity expression, we start by applying the term $q_j X_i ln(X_i)$ of equation (5.33):

$$X_{Na}ln(X_{Na}) + X_{Ca}ln(X_{Ca}) + X_{Al}ln(X_{Al}^2) + X_{Si}ln(X_{Si}^2)$$

In the *2T* model, the mole fractions of the elements are related to the mole fractions of the pure mineral members albite and anorthite by the equations:

$$X_{Na} = X_{Ab},\ X_{Ca} = X_{An},\ X_{Si} = 0.5X_{Ab},\ X_{Al} = 0.5X_{Ab} + X_{An}$$

The expression in (5.40) becomes (leaving what is inside *ln* untouched):

$$X_{Ab}ln(X_{Na}) + X_{An}ln(X_{Ca}) + (0.5X_{Ab} + X_{An})ln(X_{Al}^2) + 0.5X_{Ab}ln(X_{Si}^2)$$

Grouping terms

$$X_{Ab}ln(X_{Na}X_{Al}X_{Si}) + X_{An}ln(X_{Ca}X_{Al}^2)$$

Which is equivalent to

$$X_{Ab}ln\{Ab\}_{ideal} + X_{An}ln\{An\}_{ideal}$$

then:

$$\{An\}_{ideal} = X_{Ca}X_{Al}^2$$

$$\{Ab\}_{ideal} = X_{Na}X_{Al}X_{Si}$$

To ensure unit activity when the phase is a pure endmember we need to introduce a normalization constant. In pure albite $X_{Na} = 1$, $X_{Al} = 0.5$, and $X_{Si} = 0.5$:

$$\{Ab\}_{ideal} = 1 * 0.5 * 0.5 = 0.25$$

In pure anorthite $X_{Ca} = 1$ and $X_{Al} = 1$:

$$\{An\}_{ideal} = 1 * 1^2 = 1$$

Equations for activities with normalization constants become:

$$\{An\}_{ideal} = X_{Ca}X_{Al}^2,\ \{Ab\}_{ideal} = 4 * X_{Na}X_{Al}X_{Si}$$

Worked example 5.4

Construct a generic function to calculate solid solution ideal activities. The function should accept as parameters the phase model and composition and all needed equations to calculate proportions and activities.

This worked example shows reusable code (a general procedure) to calculate activities. Equations must be input in *Python* dictionaries and arrays with symbolic variables; for example, in the plagioclase *2T* model:

```
import sympy as sp
Abh, An, NaA, CaA, SiT, AlT = sp.symbols('Abh An NaA \
                                          CaA SiT AlT')
ca = sp.symbols('ca')
prop_eq = {AlT: 0.5*ca + 0.5, CaA: ca, NaA: 1.0 - ca,
           SiT: 0.5 - 0.5*ca, Abh: 1.0 - ca, An: ca}
act_eq = [4 * NaA * AlT * SiT, CaA * AlT**2]
```

Equations are expressed in terms of a compositional variable $ca = X^A_{Ca}$. Activities are calculated using the **subs()** function from *SymPy*. Calculated values of symbolic variables are stored in an array of tuples and are passed as arguments to the **subs()** function.The first **for** loop creates an array of tuples with compositional variables names and values for those variables. This array is then used as an argument in the **subs()** function to numerically calculate endmember and site proportions. The values of site proportions are used for substitution in the ideal activity equations (last **for** loop).

```
import numpy as np
def ideal_act(em, sites, vars, prop_eq, act_eq, comp):
    Xem = np.zeros(len(em))
    ideal_act = np.zeros(len(em))
    subs_vars = []
    for (var_name, var_value) in zip(vars, comp):
        subs_vars.append((var_name, var_value))
    subs_site_prop = []
    for site in sites:
        subs_site_prop.append((site,
                               prop_eq[site].subs(subs_vars)))
    for i in range(len(em)):
        Xem[i] = prop_eq[em[i]].subs(subs_vars)
        ideal_act[i] =  act_eq[i].subs(subs_site_prop)
    print(Xem, ideal_act)
# test the function using the plagioclase example
comp = [0.3]; vars = [ca]; em = [Abh,An]
sites = [CaA,NaA,AlT,SiT]
ideal_act(em, sites, vars, prop_eq, act_eq, comp)
```

Worked example 5.5

Derive the expressions for ideal activities and for endmember and site proportions in a muscovite model with the site distribution of Table 5.1. Use as variables:

$$y = X_{Al}^{M2A},\ x = \frac{X_{Fe}^{M2A}}{X_{Fe}^{M2A} + X_{Mg}^{M2A}}$$

Table 5.1: Cation distribution in a three-endmember muscovite

Endmember	**Formula**	**Mixing sites**				
		M2A			T1	
		Mg	Fe	Al	Si	Al
Mu	$KAl_3Si_3O_{10}(OH)_2$	0	0	1	1	1
Cel	$KMgAlSi_4O_{10}(OH)_2$	1	0	0	2	0
Fcel	$KFeAlSi_4O_{10}(OH)_2$	0	1	0	2	0

The activity expressions according to the site distribution of the model are:

$$\{Mu\}_{ideal} = 4 * X_{Al}^{M2A} * X_{Si}^{T1} X_{Al}^{T1}$$

$$\{Cel\}_{ideal} = X_{Mg}^{M2A} * (X_{Si}^{T1})^2,\ \{Fcel\}_{ideal} = X_{Fe}^{M2A} * (X_{Si}^{T1})^2$$

To find endmember (three) and site proportions (five) we are going to solve a system of eigth equations using the **solve()** function from the *SymPy* module. With two compositional variables, six more equations are needed. These equations are as follows:

- The sum of endmember proportions should be equal to one.
- Five equations, one for each cation. These are derived by taking each column in the allocation table (the number in the allocation table must be divided by the total number of cations in each site): $X_{Mg}^{M2A} = X_{Cel}$, $X_{Fe}^{M2A} = X_{Fcel}$, $X_{Al}^{M2A} = X_{Mu}$, $X_{Si}^{T1} = 1/2 * X_{Mu} + X_{Cel} + X_{Fcel}$, $X_{Al}^{T1} = 1/2 * X_{Mu}$.

```
import sympy as sp
Mu, Cel, Fcel, x, y = sp.symbols("Mu, Cel, Fcel, x, y")
AlM2A, FeM2A, MgM2A = sp.symbols("AlM2A, FeM2A, MgM2A")
SiT1, AlT1 = sp.symbols("SiT1, AlT1")
prop_eq = sp.solve([Mu + Cel + Fcel - 1,
                    AlM2A - Mu, FeM2A - Fcel, MgM2A - Cel,
                    SiT1 - 0.5 * Mu - Cel - Fcel, AlT1 - 0.5 * Mu,
                    y - AlM2A, x - FeM2A/(FeM2A+MgM2A)],
                    [Mu, Cel, Fcel, AlM2A, FeM2A,
                     MgM2A, SiT1, AlT1])
```

This code segment will result in a dictionary with equations to calculate endmember and site proportions in terms of the compositional variables:

```
{AlM2A: y, AlT1: 0.5*y, Cel: x*y-x-y+1.0, Fcel: x*(1.0-y),
 FeM2A: x*(1.0-y), MgM2A: x*y-x-y+1.0, Mu: y, SiT1: 1.0-0.5*y}
```

5.8. A *Python Class* for Ideal Solid Solution Models

So far, we have used *Python* code with procedural and functional programming techniques. *Python* can also be used with classes, using its object-oriented programming capabilities. All *Python* classes have an ***__init__()*** function executed when an object is first created (this is called *instantiation* of the class). The ***__init__()*** function is used to assign values to object properties (variables) and to perform other operations needed when an object is first created. Functions within the class are called object methods; all functions within the class take the first argument as a reference to the current instance of the class. This first argument is usually called *self*, but it does not necessarily have to be.

5.8.1. The ***__init__()*** Function

The ***__init__()*** function in the created class below (**SSPhaseIdeal** class) takes as arguments the data of the solid solution model (endmembers, crystallographic site, and equations for calculate activities, endmember proportions, and site proportions). The class has only two other functions besides the ***__init__()*** function. These two functions are used to calculate phase properties using the provided P, T, and composition. The first is a callable function (**calc_properties**) that takes as arguments the necessary data to calculate the chemical potential of endmembers. This function then calls the second function (**__gibbs**), where calculations are performed. Note that the name of the second is prefixed by two underscores; it is a usual practice to name functions that should be called only within the class this way (a private function). The separation of the functions (**calc_properties** and **__gibbs**) is made for convenience, as this class is going to be modified later.

```
import sympy
from math import exp, log
dataset = load_ds()

class SSPhaseIdeal:

    def __init__(self, endmembers, sites, vars,
                 prop_eq, ideal_act_eq):
        self.len_em = len(endmembers)
        self.endmembers = endmembers
```

(continues on next page)

(continued from previous page)

```
        self.sites = sites
        self.vars = vars
        self.prop_eq = prop_eq
        self.ideal_act_eq = ideal_act_eq

    def calc_properties(self, P, T, comp):
        self.P = P
        self.T = T
        self.comp = comp
        self.__gibbs()

    def __gibbs(self):
        R = 0.0083144626
        # Values of variables to be substituted in equations
        subs_vars = []
        for (var_name, var_value) in zip(self.vars, self.comp):
            subs_vars.append((var_name, var_value))
        Xem = np.zeros(self.len_em)  # proportions of endmembers
        for i in range(self.len_em):
            em = self.endmembers[i]
            Xem[i] = self.prop_eq[em].subs(subs_vars)
        # Values of site prop. to be substituted in activity eq.
        subs_site_prop = []
        for site in self.sites:
            subs_site_prop.append((site,
                                   self.prop_eq[site].subs(subs_vars)))
        G = 0
        Gideal = 0
        act = np.zeros(self.len_em)
        chemicalPotential = np.zeros(self.len_em)
        for i in range(self.len_em):
            em = self.endmembers[i]
            act_eq = self.ideal_act_eq[i]
            act[i] = act_eq.subs(subs_site_prop) # ideal activity
            log_act = log(act[i]) if act[i] > 0 else log(1e-64)
            # G_em in dataset:
            μo = EM_Gibbs(self.P,self.T, dataset[em.name])
            μ = μo + R*(self.T+273.15)*log_act # Mu_em in phase
            G += Xem[i] * μ                    # G_phase
            Gideal += Xem[i] * R*(self.T+273.15)*log_act
            chemicalPotential[i] = μ
        self.result = (Xem, act, chemicalPotential, Gideal, G)
```

To test the newly created class let's use the data from the last worked example:

```
Pl = SSPhaseIdeal(em, sites, vars, prop_eq, act_eq)
Pl.calc_properties(10, 500, comp)
print(Pl.result) # ([0.7, 0.3], [0.637, 0.127],
                 #  [-4060.21, -4365.80], -6.01, -4151.89)
```

5.9. Gibbs Free Energy of Non-Ideal Mixtures

This is the most general case. In non-ideal solutions, the Gibbs free energy of the phase has non-ideal contributions:

$$\overline{G}_{real} = \overline{G}_{ideal} + G_{nonideal-mix} = G_{mec} + G_{ideal-mix} + G_{nonideal-mix} \tag{5.40}$$

The Gibbs free energy of a non-ideal solid solution is then result of the Gibbs free energy of the ideal mixture (as seen in previous sections) plus the free energy coming from non-ideal interactions. The complete equation is:

$$\overline{G}_{real} = \sum_i X_i^A \mu_{i,0}^A + RT \sum_i X_i^A \ln\{i\}_{ideal}^A + RT \sum_i X_i^A \ln \lambda_i^A \tag{5.41}$$

The last term is known as the excess Gibbs free energy and is dependent on the activity coefficient λ_i^A. In the asymmetric formalism of Holland and Powell (2003), the activity coefficient is calculated from macroscopic interaction parameters (W_{ij}) using the terms in Table 5.2 and the following equation:

$$RTLn\lambda_l = -\sum_{i=1}^{n-1} \sum_{j>1}^{n} q_i q_j W_{ij}^* \tag{5.42}$$

Table 5.2: Definition of terms in the symmetric and asymmetric formalisms

Symmetric solid solutions	Asymmetric solid solutions
$q_i = 1 - X_i\ ,\ i = l$	$q_i = 1 - \phi_i\ ,\ i = l$
$q_i = -X_i\ ,\ i \neq l$	$q_i = -\phi_i\ ,\ i \neq l$
	$\phi_i = \frac{X_i \alpha_i}{\sum_{j=1}^{n} X_j \alpha_j}$
$W_{ij}^* = W_{ij}$	$W_{ij}^* = W_{ij} \frac{2\alpha_l}{\alpha_i + \alpha_j}$

The parameter W significantly affect the shape of the Gibbs free energy of mixing curve. For a more detailed analysis, see Saxena (1969).

Worked example 5.6

Calculate the non ideal activities of anorthite and albite in a plagioclase binary *2T* model. Use the following information for non ideality: $W_{abh-an} = 3.1$, $\alpha_{abh} = 0.643$, and $\alpha_{ab} = 1.0$. Use results from previous worked examples (with $ca = 0.3$) for ideal activities (0.637, 0.12675) and the equations for endmember and site proportions derive above (**prop_eq**).

To calculate activity coefficients, the following equations were derived using the asymmetric formalism:

$$\phi_{an} = \frac{X_{an} * \alpha_{an}}{X_{an} * \alpha_{an} + X_{ab} * \alpha_{ab}}$$

$$\phi_{ab} = \frac{X_{ab} * \alpha_{ab}}{X_{an} * \alpha_{an} + X_{ab} * \alpha_{ab}}$$

$$RTLn\lambda_{an} = -(1.0 - \phi_{an}) * (-\phi_{ab}) * W_{ab-an} * \frac{2 * \alpha_{an}}{\alpha_{ab} + \alpha_{an}}$$

$$RTLn\lambda_{ab} = -(1.0 - \phi_{ab}) * (-\phi_{an}) * W_{ab-an} * \frac{2 * \alpha_{ab}}{\alpha_{ab} + \alpha_{an}}$$

The following is the *Python* code implementing these equations, using **prop_eq** from the previous worked example and the asymmetric formalism-derived equations:

```
import math
X_ab = prop_eq[Ab].subs('ca', 0.3)
X_an = prop_eq[An].subs('ca', 0.3)
alpha_an = 1.0
alpha_ab = 0.643
phi_ab = X_ab*alpha_ab / (X_ab*alpha_ab+X_an*alpha_an)
phi_an = X_an*alpha_an / (X_ab*alpha_ab+X_an*alpha_an)
Waban = 3.1
RTLnlambda_an = - (1.0 - phi_an) * (- phi_ab) * Waban * \
                2 * alpha_an / (alpha_ab+alpha_an)
RTLnlambda_ab = - (1.0 - phi_ab) * (- phi_an) * Waban * \
                2 * alpha_ab / (alpha_ab+alpha_an)
R = 0.0083144626
T = 500 + 273.15
lambda_ab = math.exp(RTLnlambda_ab/(R*T)) # 1.062237
lambda_an = math.exp(RTLnlambda_an/(R*T)) # 1.235365
Real_act_ab = 0.637 * lambda_ab          # 0.677
Real_act_an = 0.12675 * lambda_an        # 0.156
```

5.10. Darken's Quadratic Formalism

Following Darken (1967), it is established that in a binary mixture, when the activity coefficient of a solvent (1) obeys the relation of a regular solution, then the activity coefficient of the solute (2) must obey the relation $RT \ln \lambda_2 = w_{12}(1-X_2)^2 + I$ where I is a constant of integration. This model is applied to a non-ideal solid solution model when the thermodynamics of an endmember in one of the terminal regions are unknown because the endmember is either fictive (not observable) or undergoes a phase transition (Powell, 1987). A simple binary solid solution illustrates the concept; as seen above, the thermodynamics of such a system can be visualized as containing three regions: the terminal regions obeying Henry's and Raoult's laws, and a transitional intermediate region that involves transitional energetic interactions.

In the Darken's quadratic formalism, in the terminal region 2, the endmembers are 1 with G_1 and a fictive endmember 2 with $G_2' = G_2 + I_2$. Then:

$$RT \ln \lambda_1 = w_{12}(1 - X_1)^2 \tag{5.43}$$

$$RT \ln \lambda_2 = I_2 + w_{12}(1 - X_2)^2 \tag{5.44}$$

where I_2 is not a function of composition; however, in general, I can be dependent on pressure and temperature ($I_i = a + bT + cP$). An example of the application of the Darken's quadratic formalism is the modeling of activity-composition relations for a plagioclase solid solution (Holland & Powell, 1992), which consists of a non-continuous series with a $C\bar{1}$ intermediate to albite-rich region and a $I\bar{1}$ anorthitic region. In the $C\bar{1}$ region, there is a real endmember (albite) and a fictive anorthite endmember with $C\bar{1}$ structure (Holland & Powell, 1992). The amphibole activity model is another example in which fictive *'make'* endmembers' properties are constrained from naturally coexisting amphiboles. Will and Powell (1992) applied the formalism to orthoamphibole, deriving the properties of an orthoedenite endmember by a linear combination of properties of edenite, anthophyllite, and cummingtonite endmembers.

5.11. The Non-Ideal *Class* for Solid Solutions

The **SSPhase** (non ideal) class is constructed by extending the functionality of the **SSPhase_ideal** class written above. The new class needs to calculate activity coefficients and take into account *Darken's quadratic formalism* so that

'fictive' endmember(s) can be considered within the class. The following are the steps to modify the **SSPhase_ideal** class:

1. Modify the ***__init__()*** function to include interaction parameters, asymmetric data, and *Darken's quadratic formalism* terms. For the interaction parameters, data should be listed considering the order of endmembers in the variable `endmembers`. For example, if endmembers are listed in the order em_1, em_2, and em_3, then, the interaction parameters should be in the order $W_{em1-em2}$, $W_{em1-em3}$, and $W_{em2-em3}$. Note also that the input interaction parameters are expected to have three terms: $W_{ij} = w_h + w_s * T + w_v * P$. The `alphas` (asymmetric) parameter must be an array with the same number of elements as the number of endmembers; if the solid solution is symmetrical, the parameter would be an array of *ones*. The *Darken's quadratic formalism* parameter (**dqf**) is an array of arrays with size zero (for real endmembers) to three (for fictive endmember).

```
    def __init__(self, endmembers, sites, vars, prop_eq, ideal_act,
                 w, alphas, dqf):
        ...
        self.w = w
        self.alphas = alphas
        self.dqf = dqf
```

2. Modify the **__Gibbs()** function to include a calculation of the summation of endmember proportions times asymmetric parameters (α), which represents the denominator of ϕ in equations from Table 5.2.

```
        Xem = np.zeros(self.len_em)    # proportions of endmembers
        sum = 0           # sum of alphas times proportions for em
        for i in range(self.len_em):
            em = self.endmembers[i]
            Xem[i] = self.prop_eq[em].subs(subs_vars)
            sum += Xem[i] * self.alphas[i]
```

3. In the **__Gibbs()** function, within the loop that calculates chemical potential of endmembers, include a **for** loop to calculate non-ideality.

```
            RTLnlambda = 0   # activity coefficient / Gexcess
            idx = 0          # counter for Ws
            for j in range(self.len_em-1):
                kroneckerJ = 1 if j==i else 0
                phiJ = self.alphas[j]*Xem[j]/sum
                for k in range(j+1, self.len_em):
                    kroneckerK = 1 if k==i else 0
                    phiK = self.alphas[k]*Xem[k]/sum
                    asf = 2*self.alphas[i]/(self.alphas[j]+ \
                                            self.alphas[k])
                    Wjk = (self.w[idx][0] + \
                           self.w[idx][1]*(self.T+273.15) + \
                           self.w[idx][2]*self.P) * asf
                    idx += 1
                    RTLnlambda += -(kroneckerJ - phiJ) *  \
                                  (kroneckerK - phiK) * Wjk
```

4. Add the non-ideality contributions to the chemical potential of the endmember and calculate the real activity.

```
            # Chemical potencial of em
            μ = μo + R*(self.T+273.15)*log_act + RTLnlambda
            # Real activity
            act[i] = act[i] * exp(RTLnlambda/(R*(self.T+273.15)))
```

Optionally, calculate the excess Gibbs free energy of the mixture and add this data to the result.

```
        Gexc = 0
        for i in range(n):
            em = self.endmembers[i]
...
            Gexc += prop[i] * RTLnlambda
```

5. Add calculations to include *Darken's quadratic formalism*.

```
...
        for i in range(self.len_em):
            em = self.endmembers[i]
            ...
            em_dqf = self.dqf[i]  # Adjust properties using DQF
```

(continues on next page)

(continued from previous page)

```
            if em_dqf:
                μo += em_dqf[0] + em_dqf[1] * (self.T+273.15) + \
                      em_dqf[2] * self.P
```

5.12. Non-Ideal Mixtures with Ordered Endmembers

In many phases, size and charge differences of the cations involved in substitutions give rise to order-disorder effects. Holland and Powell (1996) introduced the use of the symmetric formalism to deal with order-disorder in solid solutions. For example, in a solid solution involving two endmembers, where mixing occurs with ordering between two sites, the thermodynamics of the phase is described in a fictive ternary system that includes an ordered endmember and an order parameter Q (Holland & Powell, 1996). The state of order (the value of Q) can be derived using the thermodynamic data of the non-ordered endmembers and a minimization approach of the Gibbs free energy of the mixture with respect to the Q parameter.

The following are the steps to modify the **SSPhase** class to accept construction of models with ordered endmembers:

1. Import the **optimize** functions from **Scipy**

```
from scipy import optimize as opt
```

2. Add a parameter to the ***__init__()*** function of the class indicating the reactions for ordered endmembers:

```
    def __init__(self, endmembers, sites, vars, prop_eq,
                 ideal_act_eq, w, alphas, dqf, rx_ordered = None):
        self.rx_ordered = rx_ordered
```

3. Create a callback function within the class that takes the order parameters as input. The body of the function calculates $\Delta_r G$ of reactions that produce ordered endmembers. This function will be used in a root-finding algorithm (minimization) such that $\Delta_r G = 0$.

```
    # Callback function to solve for order
    def __findOrderState(self, orderParams):
        residuals = np.zeros(self.order)
        # set composition with order params
        self.comp[-self.order:] = orderParams
        self._gibbs() # calculate Gibbs
        # use reactions to calculate residuals
        for i in range (self.order):
            rx_res = 0
            rx = self.rx_ordered[i]
            for em in rx:
                idx = self.endmembers.index(em)
                coeff = rx[em]
                rx_res += coeff * self.result[2][idx]
            residuals[i] = rx_res
        return residuals
```

4. Modify the **calc_properties()** function so that, when the phase has ordered endmembers, the function calls the root-finding algorithm instead of the **__Gibbs()** function:

```
    def calc_properties(self, P, T, comp):
        self.P = P
        self.T = T
        self.comp = comp

        if self.rx_ordered: # Phase has an ordered em
            self.order = len(self.rx_ordered)
            orderVars = comp[-self.order:]
            solve_order = opt.root(self.__findOrderState,
                                   orderVars,
                                   method='lm', args=())
        else:
            self.__gibbs()
```

Note that the variable `solve_order` is not used within the class. This variable contains information about the result of the optimization (e.g., whether optimization was successful or not and the solution array). If you want to see this information, add a **print(solve_order)** statement to the code.

5. In this and other cases, solid solutions contain fictive endmembers that are constructed from a combination of existing endmembers in the dataset. The properties of these *make* endmembers are adjusted

using an approach similar to the *Darken's quadratic formalism*. To account for this scenario, we need to add yet another parameter to the ***__init__()*** function of the class and modify the code inside the main loop of the **__Gibbs()** function to calculate the properties of *make* endmembers. Note that the new parameter is a dictionary of dictionaries.

```
    def __init__(self, endmembers, sites, vars, prop_eq,
                 ideal_act_eq, w, alphas, dqf, makes = {},
                 rx_ordered = None):
        self.makes = makes
...
            # Calculate Gibbs for make endmembers
            if em in self.makes:
                make = self.makes[em]
                μo = 0
                for em_comp in make:
                    μo += EM_Gibbs(self.P,self.T,
                                   dataset[em_comp.name]) *\
                                   make[em_comp]
            else:
                # Gem in dataset
                μo = EM_Gibbs(self.P,self.T, dataset[em.name])
```

For example, in a biotite solution model, we can *make* an ordered endmember with phlogopite and annite:

```
makes = {Obi: {Phl: 2/3, Ann: 1/3}}
```

5.12.1. Final Note About the Solid Solution Class Code

The constructed class is a proof of concept, meaning it is not optimized code. In other words, the code has room for a lot of improvement (or even reorganization). For example, in the following chapter, we are going to approach the problem of Gibbs energy minimization. Currently, the **SSPhase** class recalculates the apparent Gibbs energy of endmembers every time the class method **calc_properties()** is called. It would be wise to leave the calculation of the apparent Gibbs energy of endmembers outside this method so that it is done only once during a minimization at a given pressure and temperature (when only the composition of the phase is changing).

Worked example 5.7

Plot activity-composition diagrams and Gibbs free energy vs. composition for a plagioclase *2T* solution model with albite (*Abh*) and anorthite (*An*). Also, plot the Gibbs free energy of ideal mixing and excess Gibbs free energy vs composition. Use $T = 500$ °C, $P = 10$ kbar, $Wh_{abh-an} = 3.1$, $\alpha_{ab} = 0.643$, $\alpha_{an} = 1.0$, and $I_{an} = 7.03 - 0.00466 * T$. Use X_{Ca} as the x axis. This is a model in the albite $C\bar{1}$ region, i.e., anorthite $C\bar{1}$ is a fictive endmember whose properties are adjusted from anorthite $I\bar{1}$ using the Darken's quadratic formalism.

1. With the above modifications to **SSPhase** in place we are now ready to use the non-ideal solid solution class. We call the constructor with the plagioclase solid solution model data (ideal and non-ideal).
2. Get all data in the range $X_{Ca} = 0 \ldots 1$ and plot results.

```
import numpy as np
w = [[3.1,0,0]]; alphas = [0.643, 1.0]
dqf = [[], [7.03,-0.00466,0]]
Pl = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)
Xca = np.arange(0.0,1.01,0.01); G = np.zeros(len(Xca))
Gex = np.zeros(len(Xca)); Gideal = np.zeros(len(Xca))
act_ab = np.zeros(len(Xca)); act_an = np.zeros(len(Xca))
for i in range(len(Xca)):
    Pl.calc_properties(10, 500, [Xca[i]])
    activities = Pl.result[1]
    act_ab[i], act_an[i] = activities[0], activities[1]
    Gideal[i], Gex[i] = Pl.result[3], Pl.result[4]
    G[i] = Pl.result[5]
```

Figure 5.7 shows the results of calculations.

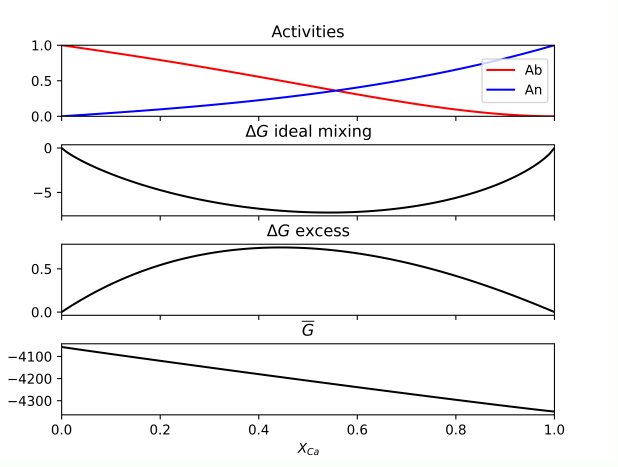

Figure 5.7: Activity, ideal Gibbs free energy of mixing, and excess Gibbs free energy vs. composition diagrams for a plagioclase solid solution ($P = 10$ kbar, $T = 500$ °C).

Worked example 5.8

Construct an activity model for a biotite solid solution in the *KFMASH* system (Table 5.3), including $Fe - Mg$ ordering, and calculate the Gibbs free energy at 10 kbar and 400C for the biotite composition: $Al^{M3} = 0.3$ and $Fe\# = 0.5$; use these two as compositional variables. Data for the symmetric non ideal solid solution are: $W_{Phl-Ann} = 12$, $W_{Phl-East} = 10$, $W_{Phl-Obi} = 4$, $W_{Ann-East} = 15$, $W_{Ann-Obi} = 8$, $W_{East-Obi} = 7$, $I_{Ann} = -3$, $I_{Obi} = -3$. The solution model must include an order parameter $Q = 3 * (Fe\# - Fe^{M12})$ and an ordered biotite endmember ($3\ Obi \leftrightarrow 2\ Phl + Ann$).

Table 5.3: Site occupancy in KFMASH Biotite

	M3			M12		T	
	Mg	Fe	Al	Mg	Fe	Si	Al
Phl	1	0	0	2	0	1	1
Ann	0	1	0	0	2	1	1
East	0	0	1	2	0	0	2
Obi	0	1	0	2	0	1	1

With the above information we can derive the following system of equations:

- $x = \frac{2*Fe^{M12}+Fe^{M3}}{2*Fe^{M12}+Fe^{M3}+2*Mg^{M12}+Mg^{M3}}$
- $y = Al^{M3}$
- $Q = 3 * (x - Fe^{M12})$
- $Mg^{M3} = phl$
- $Fe^{M3} = ann + obi$
- $Al^{M3} = east$
- $Mg^{M12} = phl + east + obi$
- $Fe^{M12} = ann$
- $Si^{T} = 0.5 * phl + 0.5 * ann + 0.5 * obi$
- $Al^{T} = 0.5 * phl + 0.5 * ann + 0.5 * obi + east$

This system can then be solved in terms of the compositional variables using the **sympy.solve()** function, as shown in previous examples, to get the equations for sites and endmembers proportions in terms of the compositional variables.

2. Use the information to construct a non-ideal solid solution and calculate properties at the specified conditions.

```
Phl, Ann, Obi, East = symbols("Phl, Ann, Obi, East")
MgM3, FeM3, AlM3 = symbols("MgM3, FeM3, AlM3")
MgM12, FeM12, SiT, AlT = symbols("MgM12, FeM12, SiT, AlT")
x, y, Q = symbols("x, y, Q")
```

Worked example 5.8 (cont.)

```
prop_eq = {Phl: -0.666666666666667*Q + x*y - x - y + 1.0,
           Ann: -0.333333333333333*Q + x,
           East: y,
           Obi: Q - x*y,
           FeM3: 0.666666666666667*Q - x*y + x,
           MgM3: -0.666666666666667*Q + x*y - x - y + 1.0,
           AlM3: y,
           MgM12: 0.333333333333333*Q - x + 1.0,
           FeM12: -0.333333333333333*Q + x,
           SiT: 0.5 - 0.5*y, AlT: 0.5*y + 0.5}
comp = [0.5, 0.3, 0]
vars = [x,y,Q]
em = [Phl, Ann, East, Obi]
sites = [FeM3, MgM3, AlM3, MgM12, FeM12, SiT, AlT]
act_eq = [4.0 * MgM3 * MgM12**2.0 * SiT * AlT,
          4.0 * FeM3 * FeM12**2.0 * SiT * AlT,
          AlM3 * MgM12**2.0 * AlT**2.0,
          4.0 * FeM3 * MgM12**2.0 * SiT * AlT]
w = [[12,0,0],[10,0,0],[4,0,0],[15,0,0],[8,0,0],[7,0,0]]
alphas = [1,1,1,1]
dqf = [[], [-3,0,0], [], [-3,0,0]]
rx_ordered = [{Obi: -1, Phl: 2/3, Ann: 1/3}]
makes = {Obi: {Phl: 2/3, Ann: 1/3}}

Bt = SSPhase(em, sites, vars, prop_eq, act_eq, w,
             alphas, dqf, makes, rx_ordered)
Bt.calc_properties(10, 400, comp)
print(Bt.result[-1], Bt.comp)        # -5946.66
                                     # [0.5, 0.3, 0.179596]
```

The last element in the composition array is the value of the order parameter Q (0.179596). Note that the input value of Q serves as an initial guess for the order parameter. In this case the value of zero was acceptable and the algorithm found a solution.

Chapter 6

Computation of Phase Diagrams

$\mu_1^A = \mu_1^B$

6.1. Introduction

This chapter presents algorithms and *Python* code to calculate phase diagrams for complex systems. Representation of phase equilibria for complex systems in two dimensions can be approached by reducing dimensions, though this comes with the inconvenient loss of information. Since pressure and temperature are commonly of interest in geology, a common approach is to construct sections of a total phase diagram (includes all the information for a system) using these two intensive variables as axes. Another commonly used approach is fixing pressure and temperature to construct compatibility (compositional) diagrams. These type of diagrams are useful, for example, in identifying reactions that could potentially occur in a system.

In some cases, it is useful to visualize all the information of the total phase diagram. This is achieved using a projection of the total phase diagram on, for example, the $P - T$ plane to show the equilibrium reactions for all possible compositions of a system. However, it is important to remember that the composition of phases can change along reactions curves, and not all reactions are relevant for a specific system composition.

Currently, the most commonly used diagrams in petrology are *pseudosections*, which are partial phase diagrams constructed for a fixed composition (or compositional range) of a rock (Hensen, 1971; Powell *et al.*, 1998). They can be built with $P - T$, $T - X$, $P - X$, or $X - X$ as axes and show only the equilibria relevant to the composition of the system. In metamorphic petrology, they are useful for documenting the metamorphic history of a rock by facilitating the construction of $P - T$ trajectories. These diagrams differ from $P - T$ projections in that they show only a few univariant reactions (in many cases, none), and most of the boundaries between assemblages indicate where a phase is consumed or produced, i.e., the mode of the phase reaches zero (mode-zero curves).

6.2. GX Diagrams and Compatibility Diagrams

As we saw with pure phases, the Gibbs free energy vs. composition diagrams (at fixed pressure and temperature) show us some of the aspects of minimizing Gibbs free energy for a system in equilibrium. The stability of assemblages can be deduced from these diagrams, allowing us to construct compatibility diagrams. Let us consider an example here of a two-component system with three phases (`A`, `B`, and `C`). In this case, the phases at equilibrium are determined by the line that is tangent to one or more

curves, such that no segment of the chemical potential curves lies below the tangent (see Figure 6.1). The representation of the stable phases in each compositional region of the system is what is known as the compatibility diagram. In this case (two-component system), it would be a straight line with segments representing the shaded regions in Figure 6.1. Note that phase C is not stable in the top diagram, where there are only three regions of phase stability: A, A+B, and B. At the pressure-temperature conditions of the bottom diagram the chemical potential for phase C has at minimum that is the lowest of the three phases and then it becomes stable; at this condition, there are five phase stability fields as shown in Figure 6.1 (boxes with stable phases).

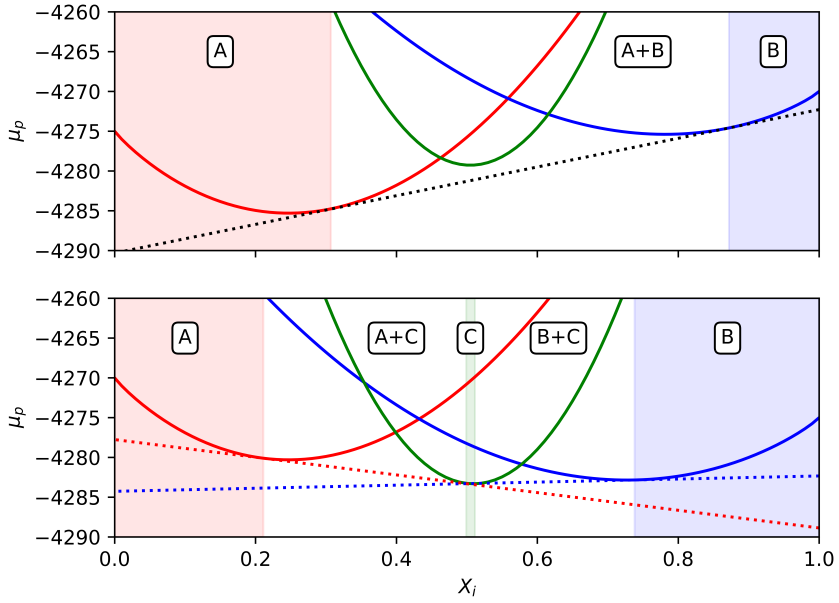

Figure 6.1: Chemical potential vs. composition diagram for a hypothetical binary system with three phases.

Worked example 6.1

Construct diagrams that show the Gibbs free energy of a alkaline feldspar solid solution versus composition at 5 kbar and 600°C, using albite and sanidine as endmembers (use a molecular mixing model). Also, plot the Gibbs free energy of mixing.

Worked example 6.1 (cont.)

First, instantiate an **SSPhase** class object with the information of the solid solution model. Then calculate properties and plot results. It is important to remember that the **SSPhase** class constructor (the ***__init__()*** method) takes several arrays and dictionary parameters with symbolic variables:

```
from sympy import symbols
Abh, San, NaA, KA, k = symbols('Abh, San, NaA, KA, k')
prop_eq = {NaA: 1.0 - k, KA: k, Abh: 1.0 - k, San: k}
vars = [k]; em = [Abh, San]; sites = [NaA, KA]
act_eq = [NaA, KA]; w = [[25.1,-0.0108,0.338]]
alphas = [1.0, 1.0]; dqf = [[], []]
AlkF = SSPhase(em,sites,vars,prop_eq,act_eq,w,alphas,dqf)
import matplotlib.pyplot as plt
var = np.arange(0,1,1/100); var[99] = 0.999999999
gibbs_ss = np.zeros(100); gibbs_mix = np.zeros(100)
P = 5; T = 600
for i in range(100):
    AlkF.calc_properties(P,T, [var[i]])
    gibbs_ss[i] = AlkF.result[5]
    gibbs_mix[i] = AlkF.result[3] + AlkF.result[4]
fig, (ax1, ax2) = plt.subplots(2, sharex=True)
ax1.plot(var, gibbs_ss, 'r'); ax2.plot(var, gibbs_mix, 'b')
```

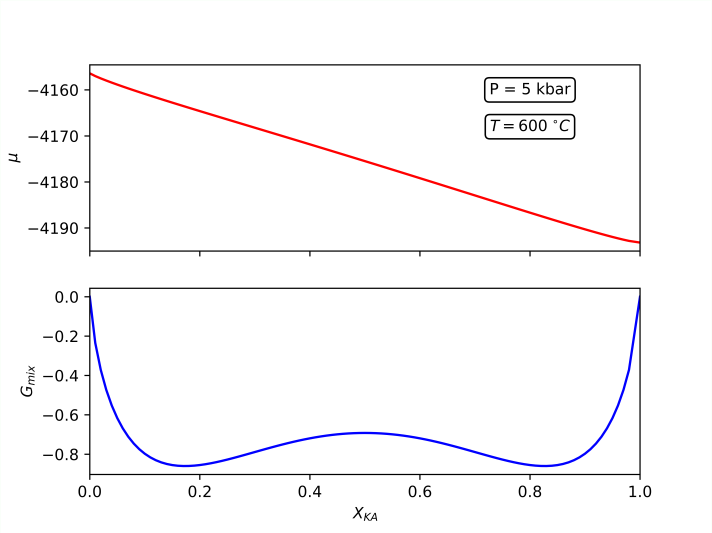

Figure 6.2: Gibbs free energy of a alkaline feldspar solid solution. Top: Total Gibbs free energy, bottom: Gibbs free energy of mixing. Binodal points are better visualized in the bottom diagram.

6.3. *TX* Diagrams

The compilation of several G vs. X diagrams at various temperatures is used to generate T vs. X diagrams. Suppose that we have generated two G vs. X diagrams in a system with three phases, with the second at a higher temperature ($T_2 > T_1$), observing the following characteristics:

- At T_1, the sequence of possible associations are from left to right: $A \rightarrow A + B \rightarrow B$.
- At some temperature between T_1 and T_2, there is a common tangent to the three G curves. This tangent line represents an equilibrium association of $A + B + C$. The sequence of possible associations are $A \rightarrow A + B + C \rightarrow B$.
- As temperature increases from low to high, the system undergoes the reaction $A + B = C$.
- At T_2, the sequence of possible associations are: $A \rightarrow A + C \rightarrow C \rightarrow B + C \rightarrow B$.

The compilation of several of this G vs. X diagrams will result in a T vs. X diagram, as shown in Figure 6.3.

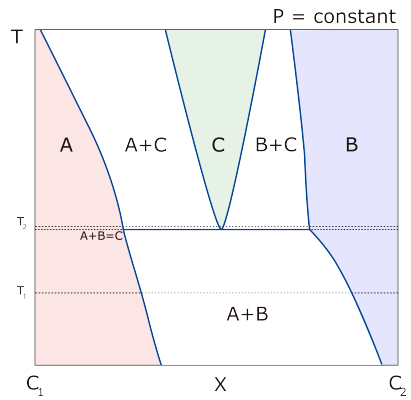

Figure 6.3: Temperature vs. composition diagram drawn from information of G vs. X diagrams.

6.4. Computation

In a system under equilibrium conditions, the calculation of phase diagrams involves finding the boundaries of equilibrium assemblages. Equilibrium

conditions are described by pressure, temperature, and the chemical potentials of components in stable phases. In a binary system, the equilibrium conditions at fixed temperature and pressure are determined by the equality of chemical potentials of components in stable phases. These chemical potentials correspond to the intersections of the dotted lines with the y axis at both extremes of the diagram in Figure 6.1. Mathematically, for the stable coexistence of phases A and B in a system with components 1 and 2, the equilibrium conditions are given by:

$$\mu_1^A = \mu_1^B \tag{6.1}$$

$$\mu_2^A = \mu_2^B \tag{6.2}$$

6.5. Phase Diagrams for Complex Systems

$P - T$ projections are constructed by finding the univariant reaction curves and their intersections. As seen in previous chapters, for pure phases (no solid solution), this problem is solved directly using the Gibbs free energy of the reactions between endmembers of the involved phases.

In systems with solid solutions, one approach to constructing phase diagrams is by solving a set of nonlinear equations for an independent set of reactions between endmembers (Powell *et al.*, 1998):

$$0 = \Delta G_n^0 + RT \ln K_n \tag{6.3}$$

In a $P - T$ diagram, the equations are solved for the composition of the phases, and for one (univariant equilibria) or both (invariant equilibria) of pressure and temperature. The same approach is also used for compatibility diagrams. The compositional coordinates of the involved phase equilibria are found by solving the set of nonlinear equations where pressure, temperature, and, for univariant equilibria, one compositional variable are fixed. For $P - T$ pseudosections, the set of nonlinear equations involves the equilibrium relationships and mass balance equations. This approach is similar to the algorithm for minimizing Gibbs free energy in systems with solid solutions.

Worked example 6.2

Draw a $T - X$ solvus diagram for the alkaline feldspar solid solution at 5 kbar and find the critical point.

First, we define an objective function that takes as parameters temperature and starting guesses for compositions of the binodals. At the binodal points the standard chemical potentials of the two endmembers are equal. This function represents a nonlinear system of two equations with two unknowns. Then the root-finding algorithm of *SciPy* is used to solve the resulting system of equations at several temperatures. The results can be used to plot the diagram (the plotting instructions are left to the reader).

```
def alkf_solvus(k, T): # P in kbar
    AlkF.calc_properties(5,T, [k[0]]); mus_k1 = AlkF.result[2]
    AlkF.calc_properties(5,T, [k[1]]); mus_k2 = AlkF.result[2]
    return mus_k2 - mus_k1
temp = np.arange(300,800,2.5); steps = len(temp)
k1 = np.zeros(steps); k2 = np.zeros(steps)
sltn_found = False
for i in range(steps):
    solution = opt.root(alkf_solvus, np.array([0.001,0.999]),
                        args=(temp[i]))
    k1[i] = solution.x[0]; k2[i] = solution.x[1]
    if not sltn_found and round(k1[i], 8) == round(k2[i],8):
        print("Critical temperature = ", temp[i])
        sltn_found = True
```

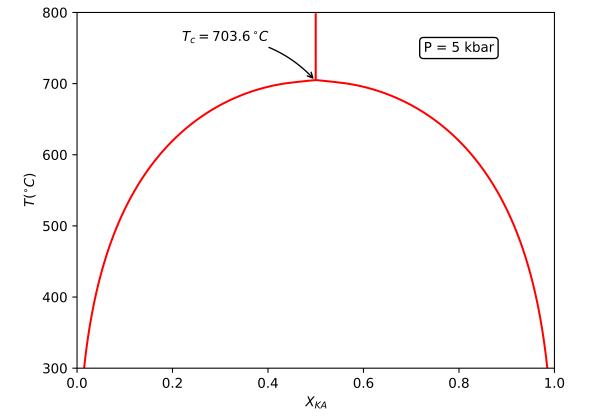

Figure 6.4: $T - X$ solvus diagram for an alkaline feldspar solid solution at 5 kbar derived from binodals in $G - X$ curves at various temperatures.

6.5.1. Gibbs Free Energy Minimization

The Gordon and McBride (1994) paper outlines the approach developed during the late 1960s at NASA to tackle general Gibbs minimization problems for large systems. The proposed algorithm uses a Taylor series expansion against $\ln n_i$ (instead of n_i), implicitly providing the positive moles constraint. Other general algorithms, like the *RAND* method, include a quadratic "Taylor development" to approximate the Gibbs free energy (Gautam & Seider, 1979). There is also a Monte Carlo-based method that does not require any approximation of the Gibbs function nor any initial guess (Liu *et al.*, 2020).

For geological applications, a number of Gibbs minimization codes have been produced over the last 30-40 years, including algorithms used in thermocalc (Powell & Holland, 1988; Powell *et al.*, 1998), Perplex (Connolly, 1990; Connolly, 2005), Theriak-Domino (de Capitani & Brown, 1987; de Capitani & Petrakakis, 2010), MELTS/pMELTS (Asimow & Ghiorso, 1998; Ghiorso, 1985; Ghiorso, 1994; Ghiorso *et al.*, 2002), MageMin (Riel *et al.*, 2022), and GeoPS (Xiang & Connolly, 2022). There are various approaches to the problem from Taylor series expansion to express the Gibbs energy of the system (MELTS and pMELTS) to solving a linearized problem with the simplex algorithm (Perple_X). In this section, we will explore the general approach to Gibbs free energy minimization, using a direct minimization and using the Lagrange multiplier method.

6.5.1.1. Direct Minimization

The goal is to minimize the function:

$$G = \sum_{i=1}^{p} n_i \mu_i \ , \tag{6.4}$$

where p is the number of phases, and n_i and μ_i are the number of moles and the chemical potential of phase i. The minimization problem is constrained by mass balance (i.e., the total amount of each component in the system must remain constant) and requiring that the number of moles of each phase remains non-negative. In summary:

$$\min G(\mathbf{n}) = \sum_{i=1}^{P} n_i \mu_i \tag{6.5}$$

$$\sum_{i=1}^{p} n_i A_{Ki} = b_K \tag{6.6}$$

$$n_i \geq 0 \tag{6.7}$$

where the subscript K represents the components of the system, (b_K is the total number of moles of component K in the system, and A_{Ki} is the number of moles of K in phase i. This optimization problem can be solved using the *SLSQP* optimization routine from the **SciPy minimize** function. However, note that the problem becomes computationally expensive for a larger number of unknowns. In the computational approach, we will define the function that calculates the Gibbs free energy of the phase assemblage and two additional functions that take care of the constraints imposed in the problem.

6.5.1.1.1 Direct Gibbs Minimization Code. We need a small change in the **SSPhase** class in the **__calc_properties** method before proceeding with the Gibbs minimization routines. The class is modified with an introduction of a new input parameter that will determine if it is necessary to calculate order state in the solid solution. This parameter will be set with a default value of **True**; this parameter is checked in the conditional statement within the method:

```
    def calc_properties(self, P, T, comp, calc_order = True):
        ...
        if self.rx_ordered and calc_order:
            ...
```

This change is necessary because the Gibbs minimization routines calculate the order state as part of the minimization process. The **__EM_Gibbs()** function is also modified to allow calculations of H_2O Gibbs free energies using the equations from Chapter 4 (see code listing in the appendix).

As a proof of concept, the script has a hard-coded system (*KFMASH*). However, this can be easily modified to make it more generic. Table 6.1 lists the compositional matrix (**A**) for the endmembers of the phases garnet, sillimanite, staurolite, white mica, biotite, chlorite, quartz, and water in *KFMASH*.

The code presented here is very inefficient since calculations of properties for solid solutions is done twice in each iteration (one in the objective function and one in the equality constraint). The following section outlines the algorithm used in the implementation.

Table 6.1: Compositional matrix of phases considered in the *KFMASH* system

em	SiO_2	Al_2O_3	FeO	MgO	K_2O	H_2O
Py	3	1	0	3	0	0
Alm	3	1	3	0	0	0
Sill	1	1	0	0	0	0
Mst	7.5	9	0	4	0	2
Fst	7.5	9	4	0	0	2
Mu	3	1.5	0	0	0.5	1
Cel	4	0.5	0	1	0.5	1
Fcel	4	0.5	1	0	0.5	1
Phl	3	0.5	0	3	0.5	1
Ann	3	0.5	3	0	0.5	1
East	2	1.5	0	2	0.5	1
Obi	3	0.5	1	2	0.5	1
Clin	3	1	0	5	0	4
Ames	2	2	0	4	0	4
Afchl	4	0	0	6	0	4
Daph	3	1	5	0	0	4
Ochl1	4	0	5	1	0	4
Ochl4	4	0	1	5	0	4
Q	1	0	0	0	0	0
H_2O	0	0	0	0	0	1

1. First, define several arrays containing the compositional matrix **A**, the **SSPhase** objects (these are found in the appendix; however, it is recommended that the reader construct the objects as an exercise), and the information to extract phase endmember compositions from **A**. Note that the code uses the previously constructed **SSPhase** class objects (variable `phases`). The variables `phase_em` and `bounds` contain the number of endmembers and the rows for endmember composition (matrix **A**) for solid solution phases.

```
A = np.array([[3,    1, 0, 3,   0, 0], [3,    1, 3, 0,   0, 0],
              [1,    1, 0, 0,   0, 0], [7.5,  9, 0, 4,   0, 2],
              [7.5,  9, 4, 0,   0, 2], [3,  1.5, 0, 0, 0.5, 1],
              [4,  0.5, 0, 1, 0.5, 1], [4,  0.5, 1, 0, 0.5, 1],
              [3,  0.5, 0, 3, 0.5, 1], [3,  0.5, 3, 0, 0.5, 1],
              [2,  1.5, 0, 2, 0.5, 1], [3,  0.5, 1, 2, 0.5, 1],
              [3,    1, 0, 5,   0, 4], [2,    2, 0, 4,   0, 4],
              [4,    0, 0, 6,   0, 4], [3,    1, 5, 0,   0, 4],
              [4,    0, 5, 1,   0, 4], [4,    0, 1, 5,   0, 4],
              [1,    0, 0, 0,   0, 0], [0,    0, 0, 0,   0, 1]])
phases = [Grt_fm,"Sill",St,WM,Bt,Chl,"Q","H2O"] # SSPhase objects
```

(continues on next page)

(continued from previous page)

```
bounds = [slice(0,2),2,slice(3,5),slice(5,8),slice(8,12),
          slice(12,18),18,19]
phase_em = [2,1,2,3,4,6,1,1]
P = 10; T = 500
```

2. Then, define an objective function (**direct_gibbs_min()**) that take as input parameters the phase compositional variables and the number of moles for phases, and return the Gibbs free energy of the system. The code also includes two constraining functions, one for mass balance and the other for constraining to positive number of moles. Note that the system composition is also hard-coded in the **mas_balance()** function: as shown in Table 6.2.

Table 6.2: Rock composition (molar)

SiO_2	Al_2O_3	FeO	MgO	K_2O	H_2O
0.369	0.053	0.041	0.021	0.017	0.5

```
def direct_gibbs_min(parameters): # parameters = [variables, moles]
    G = np.zeros(len(ass))
    N = parameters[-(len(ass)):]  # moles_phases
    counter = 0 # counter for variables
    for j in range(len(ass)):
        i = ass[j]
        N_i = N[j]
        em_i = phase_em[i]
        if em_i == 1:
                G[j] = EM_Gibbs(P,T, dataset[phases[i]])
        else:
            vars_i = parameters[counter:counter+em_i-1]
            counter += em_i-1
            phases[i].calc_properties(P,T,vars_i,False)
            G[j] = phases[i].result[5]
    return (G * N).sum()
# all numbers of moles must be ≥ 0.
def positive_moles(parameters):
    N = parameters[-(len(ass)):]
    return N
# Number of moles of each element remain constant.
def mass_balance(parameters):
    mass = np.array([0.369, 0.053, 0.041, 0.021, 0.017, 0.5])
    N = parameters[-(len(ass)):] # moles_phases
```

(continues on next page)

(continued from previous page)

```
    counter = 0 # counter for variables
    for j in range(len(ass)):
        i = ass[j]
        N_i = N[j]
        em_i = phase_em[i]
        A_i = np.array(A[bounds[i]])
        if em_i == 1:
            mass_balance_R = mass_balance_R - N_i * A_i
        else:
            vars_i = parameters[counter:counter+em_i-1]
            counter += em_i-1
            phases[i].calc_properties(P,T,vars_i,False)
            X_em = phases[i].result[0]
            mass = mass - N_i * np.dot(A_i.T, X_em)
    return mass
```

Worked example 6.3

Use the previously written code for direct Gibbs minimization code to find Gibbs free energy of the system and the composition of the phases at the minimum of the Gibbs free energy surface at $P = 10\,\text{kbar}$, $T = 618\,°C$ and containing $Grt - WM - Qz - H_2O$.

```
from scipy.optimize import minimize
x_grt = 0.6; x_wm = 0.5; y_wm = 0.3
Ngrt = 0.015; Nwm = 0.03; Nq = 0.2; Nh2o = 0.45
x0 = np.array([x_grt, x_wm, y_wm, Ngrt, Nwm, Nq, Nh2o])
ass = [0,3,6,7] #grt, wm, q, h2o,
sol = minimize(
    direct_gibbs_min, x0, method='SLSQP',
    constraints=[
        {'type': 'eq','fun': mass_balance},
        {'type': 'ineq','fun': positive_moles}
        ],
    options={'maxiter': 1000}
    )
```

This routine will successfully calculate Gibbs free energy for the considered assemblage ($Grt - WM - Qz - H_2O$) and will produce the following results: $x_{Grt} = 0.78$, $x_{WM} = 0.25$, $y_{WM} = 0.59$, $N_{Grt} = 0.016$, $N_{WM} = 0.034$, $N_Q = 0.205$, $N_{H_2O} = 0.466$ and Gibbs of the system = -647.488 kJ. Note that not restricting the values of variables will lead the code to crash for invalid values within exponentials in the Gibbs function (try, for example, calculations including chlorite).

6.5.1.2. Minimization of Gibbs Free Energy with Lagrange Multipliers

6.5.1.2.1 Lagrange Multipliers. The method of Lagrange multipliers is introduced with an example. Suppose we want to minimize the function $f(x)$ with constraints imposed by a second function $h(x)$:

$$f(x) = x_1 + x_2 \tag{6.8}$$

$$h(x) = x_1^2 + x_2^2 - 1 = 0 \tag{6.9}$$

The problem is approached by constructing a *Lagrangian* function, which results from the addition of the function $f(x)$ with its constraints multiplied by some factors called *Lagrange multipliers*, symbolized as Λ:

$$L(x, \Lambda) = f(x) + \Lambda h(x) \tag{6.10}$$

$$L(x, \Lambda) = x_1 + x_2 + \Lambda(x_1^2 + x_2^2 - 1) \tag{6.11}$$

Deriving this function with respect to x_1, x_2, and Λ, we obtain:

$$\frac{\partial L}{\partial x_1} = \frac{\partial f}{\partial x_1} + \Lambda \frac{\partial h}{\partial x_1} = 1 + 2\Lambda x_1 = 0 \tag{6.12}$$

$$\frac{\partial L}{\partial x_2} = \frac{\partial f}{\partial x_2} + \Lambda \frac{\partial h}{\partial x_2} = 1 + 2\Lambda x_2 = 0 \tag{6.13}$$

$$\frac{\partial L}{\partial \Lambda} = \frac{\partial \Lambda h}{\partial \Lambda} = h = x_1^2 + x_2^2 - 1 = 0 \tag{6.14}$$

This is a system of three equations with three unknowns (the two variables of the function and a Lagrange multiplier), which can be solved using conventional methods.

```
def lagrangian_fx(v):
    (x1,x2,l) = v
    R1 = 1 + 2*l*x1
    R2 = 1 + 2*l*x2
    R3 = x1*x1 + x2*x2 - 1
    return np.array([R1,R2,R3])
solution = opt.root(lagrangian_fx, [-1,-1,0])
print(solution)
```

```
message: The solution converged.
success: True
 status: 1
    fun: [ 9.564e-12  2.120e-11 -1.553e-11]
      x: [-7.071e-01 -7.071e-01  7.071e-01]
   nfev: 18
   fjac: [[-7.065e-01 -2.575e-02  7.072e-01]
          [ 4.295e-01 -8.098e-01  3.996e-01]
          [-5.625e-01 -5.861e-01 -5.832e-01]]
      r: [-2.042e+00 -1.030e+00  1.047e+00 -1.704e+00
           5.495e-01 1.629e+00]
    qtf: [-5.414e-09 -5.504e-09 -3.359e-09]
```

6.5.1.2.2 Non-Stoichiometric Formulation of the Minimization Problem. The non-stoichiometric formulation treats the closed-system constraint in the minimization problem using Lagrange multipliers (Smith & Missen, 1982). The problem consists of minimizing G for fixed conditions of P and T, subject to the restrictions dictated by mass conservation equations, as in the direct minimization approach.

We have M conservation equations, one for each K component in the system. Note that $M = \text{rank}(\mathbf{A})$, where $\mathbf{A}$ is the compositional matrix of the system. The conservation equations can be written as:

$$b_K - \sum_{i=1}^{N} n_i A_{Ki} = 0 \tag{6.15}$$

where N is the number of phases. The Lagrange multiplier method allows the removal of the closed-system constraint. The Lagrangian function of the problem is expressed as:

$$L(\mathbf{n}, \Lambda) = \sum_{i=1}^{N} n_i \mu_i + \sum_{K=1}^{M} \Lambda_K \left(b_K - \sum_{i}^{N} n_i A_{Ki} \right) \tag{6.16}$$

where Λ is a vector of M unknown Lagrange multipliers (one for each component in the system). The conditions of the problem result in a set of $(N + M)$ equations with $(N + M)$ unknown variables $(n_1, \ldots, n_N, \Lambda_1, \ldots, \Lambda_M)$:

$$\left(\frac{\partial L}{\partial n_i} \right)_{n_{j \neq i}, \Lambda} = \mu_i - \sum_{K=1}^{M} \Lambda_K A_{Ki} = 0 \tag{6.17}$$

$$\left(\frac{\partial L}{\partial \Lambda_K} \right)_{\mathbf{n}, \Lambda_{J \neq K}} = b_K - \sum_{i=1}^{N} n_i A_{Ki} = 0 \tag{6.18}$$

These equations result from the application of the *Kuhn-Tucker* theorem for optimization problems, which asserts that the optimal point (equilibrium) for a given function under constraints (system composition) can be found if the gradient in the target function can be expressed as a definite linear combination of normals to the constraint surfaces. These equations, however, only provide the necessary conditions; the found solution must be checked for all limiting directions of feasible assemblages in order to check if it is indeed a global minimum. Each Λ_K represents the chemical potential of the respective component. In fact, at equilibrium, the Gibbs free energy of the system can be calculated from:

$$G = \sum_{K=1}^{M} \Lambda_K b_K \tag{6.19}$$

The nonlinearity of the Gibbs free energy function in a solid solution complicates the problem above, as finding a solution requires searching for the stable phases present and their compositions.

Worked example 6.4

Formulate the equations for Gibbs minimization using Lagrange multipliers for an assemblage of $Grt - WM - Qz - H_2O$ in the *KFMASH* system, using the rock composition from worked example 6.3. You need to formulate one equation for each endmember in the solid solutions and one equation for each pure phase. There are six lagrange multipliers, one for each component:

$$\Lambda = [\Lambda_{SiO_2}, \Lambda_{Al_2O_3}, \Lambda_{FeO}, \Lambda_{MgO}, \Lambda_{K_2O}, \Lambda_{H_2O}]$$

---

1. Equations for chemical potentials and Lagrange multipliers:

$$\mu_{Py}^{Grt} - (3\Lambda_{SiO_2} + \Lambda_{Al_2O_3} + 3\Lambda_{MgO}) = 0$$

$$\mu_{Alm}^{Grt} - (3\Lambda_{SiO_2} + \Lambda_{Al_2O_3} + 3\Lambda_{FeO}) = 0$$

$$\mu_{Mu}^{WM} - (3\Lambda_{SiO_2} + 1.5\Lambda_{Al_2O_3} + 0.5\Lambda_{K_2O} + \Lambda_{H_2O}) = 0$$

$$\mu_{Cel}^{WM} - (4\Lambda_{SiO_2} + 0.5\Lambda_{Al_2O_3} + \Lambda_{MgO} + 0.5\Lambda_{K_2O} + \Lambda_{H_2O}) = 0$$

$$\mu_{Fcel}^{WM} - (4\Lambda_{SiO_2} + 0.5\Lambda_{Al_2O_3} + \Lambda_{FeO} + 0.5\Lambda_{K_2O} + \Lambda_{H_2O}) = 0$$

$$\mu_{Q}^{Qz} - \Lambda_{SiO_2} = 0$$

$$\mu_{H_2O}^{Fluid} - \Lambda_{H_2O} = 0$$

Worked example 6.4 (cont.)

These equations result from a matrix operation. For example, for garnet, the equations are represented by:

$$A_{Grt} = \begin{bmatrix} 3 & 1 & 0 & 3 & 0 & 0 \\ 3 & 1 & 3 & 0 & 0 & 0 \end{bmatrix}$$

$$[\mu_{Py}, \mu_{Alm}] - (A_{Grt} \cdot \Lambda)$$

2. Mass balance equations:

$$\begin{aligned} SiO2_2 :\ & 0.369 - N^{Grt}3(X_{Py} + X_{Alm}) \\ & - N^{WM}(3X_{Mu} + 4X_{Cel} + 4X_{Fcel}) \\ & - N^{Qz} = 0 \end{aligned}$$

$$\begin{aligned} Al_2O_3 :\ & 0.053 - N^{Grt}(X_{Py} + X_{Alm}) \\ & - N^{WM}(1.5X_{Mu} + 0.5X_{Cel} + 0.5X_{Fcel}) = 0 \end{aligned}$$

$$FeO :\ 0.041 - N^{WM}X_{Fcel} - N^{Grt}3X_{Alm} = 0$$

$$MgO :\ 0.021 - N^{WM}X_{Cel} - N^{Grt}3X_{Py} = 0$$

$$K_2O :\ 0.017 - N^{WM}0.5(X_{Mu} + X_{Cel} + X_{Fcel}) = 0$$

$$H_2O :\ 0.5 - N^{WM}(X_{Mu} + X_{Cel} + X_{Fcel}) - N^{H2O} = 0$$

These equations can be represented with matrix operations. For example, the equivalent operation for garnet is:

$$X_{em}^{Grt} = [X_{Py}, X_{Alm}]$$

$$MassBalance = MassBalance - N^{Grt} * \left(A_{Grt}^{T} \cdot X_{em}^{Grt}\right)$$

6.5.1.2.3 A Python Class for Performing Gibbs Minimization with Lagrange Multipliers. This class uses previously constructed **SSPhase** class objects, and so the class has the following hard-coded array:

```
self.phases = [Grt_fm,"Sill",St,WM,Bt,Chl,"Q","H2O"]
```

The following is the algorithm:

1. In the ***__init__()*** method, define an array containing the class objects with considered phases in the system, the compositional matrix, and some useful variables to recover endmember compositional information from the compositional matrix (number of endmembers in each phase and the row position of the endmember compositions in matrix **A**). Note that the hard-coded compositional matrix (**A**) can

be alternatively constructed using the imported dataset and the information contained in the solid solution phase class objects.

2. Write an objective function within the class to find the equilibrium condition (minimization) with Lagrange multipliers. The function will accept as parameters the compositional variables in the solid solutions, the number of moles for phases, and the Lagrange multipliers. It returns an array with residuals from applying the Lagrange multipliers algorithm as explained above. The objective function uses a utility function to calculate phase properties; this function stores the properties in arrays available to the class.

3. There is one callable method to perform calculations with a provided assemblage at some set of pressure and temperature conditions. Within this function, the root-finding routine is called with the objective function as an argument.

The class is named **PhaseEquilibria**, and its functionality will be extended during the rest of this chapter. The following is the *Python* code:

```
import numpy as np
import scipy.optimize as opt
class PhaseEquilibria:
    def __init__(self, rock_comp = None):
        self.A = np.array([[3,    1, 0, 3,   0, 0],
                           [3,    1, 3, 0,   0, 0],
                           [1,    1, 0, 0,   0, 0],
                           [7.5,  9, 0, 4,   0, 2],
                           [7.5,  9, 4, 0,   0, 2],
                           [3,  1.5, 0, 0, 0.5, 1],
                           [4,  0.5, 0, 1, 0.5, 1],
                           [4,  0.5, 1, 0, 0.5, 1],
                           [3,  0.5, 0, 3, 0.5, 1],
                           [3,  0.5, 3, 0, 0.5, 1],
                           [2,  1.5, 0, 2, 0.5, 1],
                           [3,  0.5, 1, 2, 0.5, 1],
                           [3,    1, 0, 5,   0, 4],
                           [2,    2, 0, 4,   0, 4],
                           [4,    0, 0, 6,   0, 4],
                           [3,    1, 5, 0,   0, 4],
                           [4,    0, 5, 1,   0, 4],
                           [4,    0, 1, 5,   0, 4],
                           [1,    0, 0, 0,   0, 0],
```

(continues on next page)

(continued from previous page)

```
                        [0,    0, 0, 0,   0, 1]])
        self.phases = [Grt_fm,"Sill",St,WM,Bt,Chl,"Q","H2O"]
        self.phases_em = np.array([2,1,2,3,4,6,1,1])
        self.bounds = [slice(0,2),slice(2,3),slice(3,5),slice(5,8),
                       slice(8,12),slice(12,18),slice(18,19),
                       slice(19,20)]
        self.rock_comp = rock_comp

    def calculate(self, calc_type, assemb, param, intensive_vars):
        self.assemb = assemb
        self.total_em = self.phases_em[assemb].sum()
        self.G = np.zeros(len(assemb))
        # construct A_assemb
        ix_em = np.hstack([np.ogrid[self.bounds[p]] \
                           for p in assemb])
        A_assemb = self.A[ix_em.astype(int)]
        self.A_assemb = A_assemb
        #============================Gibbs minimization block
        if calc_type == "Gibbs_min":
            self.P = intensive_vars[0]
            self.T = intensive_vars[1]
            # add Lagrange multipliers
            param_ = np.r_[param, np.full(6,-1000.)]
            opt_result = opt.root(self.__Lagrangian, param_)
            # moles of phases
            moles = opt_result.x[0:(len(self.assemb))]
            vars = opt_result.x[len(self.assemb):-6]
            lagrange_mult = opt_result.x[-6:]
            gibbs = np.dot(self.G, moles)
            self.result = (moles, vars, lagrange_mult, gibbs)

    #============================= Function for Gibbs Minimization
    def __Lagrangian(self, param):
        moles = param[0:(len(self.assemb))]  # moles_phases
        vars = param[len(self.assemb):-6]
        lag_mul = param[-6:]
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_L = self.μ - np.dot(self.A_assemb, lag_mul)
        return np.r_[r_mb, r_L]
    #====================== end of Function for Gibbs Minimization

    #============================= utility Functions
```

(continues on next page)

(continued from previous page)

```
    def __calcPhaseProperties(self, vars):
        self.μ = np.zeros(self.total_em)
        self.X_em = np.zeros(self.total_em)
        self.phases_comp = np.zeros((len(self.assemb), 6))
        ix_em = 0  # index for first endmember in phase i
        for i in range(len(self.assemb)):
            p = self.assemb[i]
            em = self.phases_em[p]  # number of endmembers in phase
            A_i = self.A[self.bounds[p]]
            if em == 1:
                self.μ[ix_em] = EM_Gibbs(self.P,self.T,
                                         dataset[self.phases[p]])
                self.G[i] = self.μ[ix_em]
                self.X_em[ix_em] = 1.0
                self.phases_comp[i,:] = A_i # comp. of pure phase
            else:
                ix_var = ix_em - i
                p_vars = vars[ix_var:ix_var+em-1]
                self.phases[p].calc_properties(self.P,self.T,
                                               p_vars,False)
                self.μ[ix_em:ix_em+em] = self.phases[p].result[2]
                X_em_p = self.phases[p].result[0]
                self.X_em[ix_em:ix_em+em] = X_em_p
                self.G[i] = self.phases[p].result[5]
                # comp. of ss phase
                self.phases_comp[i,:] = np.dot(A_i.T, X_em_p)
            ix_em += em
```

Worked example 6.5

Use the **PhaseEquilibria** class to find equilibrium composition of phases and Gibbs free energy of the rock system used previously. Test various sets of assemblages.

---

1. We start with: $Grt + WM + Qz + H_2O$.

```
minim = PhaseEquilibria(rock_comp = np.array([0.369, 0.053,
                                0.041, 0.021, 0.017, 0.5]))
x_grt = 0.6; x_wm = 0.5; y_wm = 0.3
Ngrt = 1; Nwm = 1; Nq = 1; Nh2o = 1
x = np.array([Ngrt, Nwm, Nq, Nh2o, x_grt, x_wm, y_wm])
minim.calculate("Gibbs_min", np.array([0,3,6,7]), x, [10.,500.])
(moles, vars, lagrange_mult, gibbs) = minim.result
```

Worked example 6.5 (cont.)

This will produce similar results as with the direct method: variables = [0.78 0.26 0.59], N = [0.016, 0.034, 0.205, 0.466], Gibbs: -647.488 kJ.

2. Do a Gibbs minimization with the assemblage $Grt + WM + Bt + Qz + H_2O$.

```
x_grt = 0.6; x_wm = 0.5; y_wm = 0.3
x_bi = 0.5; y_bi = 0.9; Q_bi = 0.02
Ngrt = 0.015; Nwm = 0.03; Nbi = 0.01; Nq = 0.2; Nh2o = 0.45
x = np.array([Ngrt, Nwm, Nbi, Nq, Nh2o, x_grt, x_wm, y_wm,
              x_bi, y_bi, Q_bi])
minim.calculate("Gibbs_min", np.array([0,3,4,6,7]),
                x, [10.,500.])
(moles, vars, lagrange_mult, gibbs) = minim.result
```

This will produce: variables = [0.83, 0.33, 0.87, 0.43, 0.13, 0.167], N = [0.012, 0.026, 0.008, 0.228, 0.466], Gibbs = -647.560 kJ. This is a more stable assemblage than $Grt + WM + Qz + H_2O$. Note that the calculate order variable Q_{Bt} (0.167) is the same as obtained by doing:

```
Bt.calc_properties(10, 500, [0.43, 0.13, 0.01])
```

3. Left to the reader: do calculations with other assemblages and observe cases where the solution includes negative moles for phases, variable values outside range, and crashing of the algorithm because of invalid values in exponentials.

6.5.1.3. Tests for Stability of Phases at Equilibrium

6.5.1.3.1 Criteria Using the Kuhn-Tucker Conditions. The application of this criterion consists of determining whether a given phase possesses an energy that is minimal compared to all other phases considered in the current calculation. Following this reasoning, the criteria to determine if phases should be considered in equilibrium are (Smith & Missen, 1982):

- For pure phases:

$$\mu_{i,0} - \sum_{K=1}^{M} \Lambda_K A_{Ki} = 0, \quad (n_i > 0) \tag{6.20}$$

$$\mu_{i,0} - \sum_{K=1}^{M} \Lambda_K A_{Ki} > 0, \quad (n_i = 0) \tag{6.21}$$

The second equation implies that the formation of a phase is not possible if the Gibbs free energy of the system increases during this process. This is the Kuhn-Tucker condition, which applies to absent phases.

- For ideal solid solutions:

$$\mu_i = \mu_{i,0} + RT \ln\{i\}_{ideal} = \sum_{K=1}^{M} \Lambda_K A_{Ki} \tag{6.22}$$

$\mu_{i,0}$ is the standard chemical potential at the pressure and temperature of interest of the endmember, and $\{i\}_{ideal}$ is the ideal activity of the endmember in the solid solution.

From this equation, we obtain:

$$\{i\}_{ideal} = e^{\left[\frac{1}{RT}\left(-\mu_{i,0}+\sum_{K=1}^{M}\Lambda_K A_{Ki}\right)\right]} \tag{6.23}$$

If $\sum_i \{i\}_{ideal} < 1$ the phase is not stable, if $\sum_i \{i\}_{ideal} = 1$ the phase is in incipient formation; and if $\sum_i \{i\}_{ideal} > 1$ the phase is present and must be considered. The criterion must also be applied to the base phases to know if they should be changed to free phases.

If the solid solution is non-ideal, the equivalent equation is:

$$\lambda_i \{i\}_{ideal} = e^{\left[\frac{1}{RT}\left(-\mu_{i,0}+\sum_{K=1}^{M}\Lambda_K A_{Ki}\right)\right]} \tag{6.24}$$

Worked example 6.6

Test sillimanite and staurolite stability against the assemblage $Grt + WM + Bt + Qz + H_2O$ (worked example 6.5) using the criteria from the Kuhn-Tucker conditions.

1. Testing sillimanite and staurolite stability:

```
# Sillimanite
lag_mul =  np.array([-934.1790866, -1718.60876431,
                     -332.44024278, -663.18266589,
                     -890.42412431, -338.13545873])
print(EM_Gibbs(10,500, dataset["Sill"]) - \
      np.dot(np.array([1, 1, 0, 0, 0, 0]), lag_mul))
# Staurolite
import math
R = 0.0083144626; P = 10; T = 500
mu_o_mst = EM_Gibbs(P,T, dataset["Mst"])
mu_o_fst = EM_Gibbs(P,T, dataset["Fst"])
real_act_mst = math.exp(1/(R*(T+273.15))*(-mu_o_mst + \
    np.dot(np.array([7.5,  9,  0,  4,  0,  2]), lag_mul)))
real_act_fst = math.exp(1/(R*(T+273.15))*(-mu_o_fst + \
    np.dot(np.array([7.5,  9,  4,  0,  0,  2]), lag_mul)))
print(real_act_mst + real_act_fst)
```

Worked example 6.6 (cont.)

For sillimanite, the script gives a value greater than zero, meaning that sillimanite is not stable and should not be considered in further calculations. For staurolite, we get a value greater than one and therefore the phase should be considered.

2. Let us test the new assemblage $Grt + St + WM + Bt + Qz + H_2O$ (Note the change in starting guesses):

```
x_grt = 0.8; x_st = 0.7; x_wm = 0.3; y_wm = 0.8;
x_bi = 0.4; y_bi = 0.1; Q_bi = 0.05
Ngrt = 0.01; Nst = 0.02; Nwm = 0.02;
Nbi = 0.001; Nq = 0.2; Nh2o = 0.45
x = np.array([Ngrt, Nst, Nwm, Nbi, Nq, Nh2o, x_grt, x_st,
              x_wm, y_wm, x_bi, y_bi, Q_bi])
minim.calculate("Gibbs_min", np.array([0,2,3,4,6,7]),
                x, [10.,500.])
(moles, vars, lagrange_mult, gibbs) = minim.result
```

This calculation is successful and returns $G = -647.568$ kJ, a value below the Gibbs free energy of the assemblage without staurolite ($G = -647.560$ kJ).

6.5.1.3.2 Criteria Using the "Tangent Plane Distance Function" (TPDF). In the previous criteria for solid solutions, sum values less than one result from negative exponents, meaning that the standard chemical potentials for all endmembers lie above the hyperplane. According to this criterion, the phase should not be considered in calculations. However, this test does not a guarantee that, at some composition of the solid solution, the Gibbs free energy of the phase lies below the hyperplane. A better stability test should look for bumps lying below that plane. This is where the "Tangent Plane Distance Function" or *TPDF* comes in handy. The *TPDF* determines how far the Gibbs surface is from the tangent hyperplane found during minimization. For a detailed derivation of the *TPDF*, see Michelsen (1982). To identify missing phases, this test relies on finding whether there are phases where the molar Gibbs energy of the solid solution ever falls below the tangent hyperplane found during the Gibbs minimization. To test a trial phase with composition `x`:

$$TPDF = G_{ss,x} - G_{hyperplane,x} = \sum_{i=1}^{em} X_{i,x}(\mu_{i,x} - \sum_{K=1}^{M} \Lambda_K A_{K,i}) \tag{6.25}$$

In matricial form:

$$TPDF = G_{ss,x} - \mathbf{X_{em}} \cdot \mathbf{A_{phase}} \cdot \Lambda^T \tag{6.26}$$

$\mu_{i,x}$ and X_i are the chemical potential and mole fraction of endmember i in the considered phase at x and $A_{K,i}$ is the content of component K in the endmember.

The distance is determined by minimizing the *TPDF*. To guarantee a successful search, the minimization algorithm needs as constraints the valid range for the compositional variables. If the minimizer returns a negative value for the *TPDF*, then the tangent hyperplane intersects the Gibbs energy surface of the considered phase. This means the phase must be included in the stable assemblage.

Worked example 6.7

Check that the found tangent hyperplane is a solution to the Gibbs minimization problem at the found biotite composition in the assemblage $Grt + WM + Bt + Qz + H_2O$. Check staurolite stability against the assemblage $Grt + WM + Bt + Qz + H_2O$ using the *TPDF*.

---

1. Calculate properties of biotite and get proportion of endmembers and the Gibbs energy of the solid solution at the found biotite composition (x).
2. Use the found Lagrange multipliers and the biotite compositional matrix together with the endmember proportions in the *TPDF*.

```
Bt.calc_properties(10, 500, [0.423, 0.13, 0.01])
X_em = Bt.result[0]
Gbt_x = Bt.result[5]
lag_mul =  np.array([-934.1790866, -1718.60876431,
                     -332.44024278,-663.18266589,
                     -890.42412431, -338.13545873])
A_bt = np.array([[3,0.5,0,3,0.5,1], [3,0.5,3,0,0.5,1],
                 [2,1.5,0,2,0.5,1], [3,0.5,1,2,0.5,1]])
G_plane = np.dot(X_em, np.dot(A_bt, lag_mul.T))
print(round(Gbt_x - G_plane, 6))
```

This script will return a value of zero indicating that effectively the hyperplane is tangent to the Gibbs free energy surface of biotite at x.

3. Create an objective *Python* function for the *TPDF* that takes one parameter (a vector containing the value of the compositional variable). The function: (i) calculates the properties of staurolite (endmember proportions and solid solution Gibbs energy at **x**), (ii) calculate Gibbs energy of the hyperplane at **x** (the dot products in the *TPDF* equation), (iii) returns the difference between the Gibbs free energy of staurolite at **x** and the Gibbs energy of the hyperplane at **x**.
4. Create two auxiliary functions to handle the inequality constraint $0 \leq \mathbf{x} \leq 1$
5. Call the **minimize** function from the **scipy.optimize** module with an starting guess for **x**.

Worked example 6.7 (cont.)

```
def st_TPDF(x):
    St.calc_properties(10, 500, x)
    Xem_x = St.result[0]
    Gst_x = St.result[5]
    lag_mul =  np.array([-934.1790866, -1718.60876431,
                         -332.44024278, -663.18266589,
                         -890.42412431, -338.13545873])
    A_st = np.array([[7.5,  9,  0,  4,  0,  2],
                     [7.5,  9,  4,  0,  0,  2]])
    G_plane_x = np.dot(Xem_x, np.dot(A_st, lag_mul.T))
    return Gst_x - G_plane_x

def gtz(x): # greater than zero
    return x

def lto(x): # less than one
    return x - 1

from scipy.optimize import minimize
x0 = [0.9]
sol = minimize(
    st_TPDF, x0, method='SLSQP',
        constraints=[
        {'type': 'ineq','fun': gtz},
        {'type': 'ineq','fun': lto}
        ],
    options={'maxiter': 1000}
    )
print(sol.fun)
```

The return value (-6.8964) is negative indicating that staurolite should be considered as a stable phase. Note that this minimum occurs at $\mathbf{x} = [1]$, i.e., at one of the extremes of the staurolite solid solution. If we had not used the inequality constraints, the routine would have crashed for the value of $\mathbf{x}$ would go outside its valid range. This outcome was expected because we knew that the standard chemical potential of one endmember lies below the tangent hyperplane as indicated in the test with the Kuhn-Tucker conditions.

6.5.2. Construction of Phase Diagrams Using a System of Nonlinear Equations

This is the approach used by Powell and Holland (1988) and Powell *et al.* (1998), the algorithm begins by identifying an independent set of reactions between endmembers of the considered assemblage.

For each reaction, there will be a $\Delta_r G$ equation, which, under equilibrium conditions, is written as:

$$0 = \Delta_r G^0 + RT \ln K \tag{6.27}$$

If we have a reaction:

$$xA + yB \leftrightarrow uC + vD \tag{6.28}$$

The equilibrium condition is written as:

$$0 = u\mu_C^0 + v\mu_D^0 - (x\mu_A^0 + y\mu_B^0) + RT \ln \frac{\{C\}^u \{D\}^v}{\{A\}^x \{B\}^y} \tag{6.29}$$

reorganizing:

$$0 = u(\mu_C^0 + RT \ln\{C\}) + v(\mu_D^0 + RT \ln\{D\}) - (x(\mu_A^0 + RT \ln\{A\}) + y(\mu_B^0 + RT \ln\{B\})) \tag{6.30}$$

or:

$$0 = u\mu_C + v\mu_D - (x\mu_A - y\mu_B) \tag{6.31}$$

Generalizing this expression, we obtain:

$$0 = \sum_{products} rc_i \mu_i - \sum_{reactants} rc_j \mu_j \tag{6.32}$$

where rc_i and rc_j are reaction coefficients, and μ_i and μ_j are the chemical potentials of the endmembers in their respective phases.

This calculation can be performed using linear algebra operation. Equation (6.32) is equivalent to the dot product:

$$0 = \mathbf{R} \cdot \mu \tag{6.33}$$

where matrix $\mathbf{R}$ contains the coefficients (rc) of independent reactions:

$$\mathbf{R} = \begin{bmatrix} rc_{1,1} & rc_{1,2} & \dots & rc_{1,em_T} \\ rc_{2,1} & rc_{2,2} & \dots & rc_{2,em_T} \\ \dots & & & \\ rc_{n_{ir},1} & rc_{n_{ir},2} & \dots & rc_{n_{ir},em_T} \end{bmatrix} \tag{6.34}$$

Here, em_T represents the total number of endmembers (across all considered phases), $n_{ir} = em_T - c$ is the number of independent reactions, and c

is the rank of the compositional matrix (A; composition of endmembers). The vector μ corresponds to the chemical potential of the endmembers:

$$\mu = [\mu_1, \ldots, \mu_T] \tag{6.35}$$

The number of independent reactions in a c-component system depends on the total number of endmembers that make up the number of phases (n_{phases}) in the equilibrium under consideration. Because each phase is defined by having $n_{em}^{phase} - 1$ compositional variables (unknowns), there will be a total of $n_{em}^{T} - n_{phases} + 2$ unknowns in the problem. The number of phases dictates how many variables need to be set in order to find the equilibrium conditions, in accordance with the phase rule. For example:

- If there are $c + 2$ phases, the equilibrium is invariant (a point in a pressure-temperature diagram).
- If there are $c + 1$ phases, the equilibrium is univariant. In a pressure-temperature projection, one variable (pressure or temperature) needs to be set to find the equilibrium conditions.
- If there are c phases, the equilibrium is divariant. In a compatibility diagram, for example, if pressure and temperature are fixed, then the composition of the coexisting phases under equilibrium can be calculated.

For a more detailed discussion of possibilities, see Powell *et al.* (1998).

6.5.2.1. Calculating Reaction Curves in Pressure-Temperature Projections (*Petrogenetic Grids*)

The *petrogenetic grid* term was coined by Bowen (1940), referring to intersecting reaction curves cutting a $P - T$ diagram into a grid. The reaction curves bound all possible divariant equilibria in $P - T$ space for a specific compositional system. In $P - T$ projections, invariant equilibria is characterized by having $c + 2$ phases and therefore there are n equations with the same number of unknowns; the equilibria is a point in $P - T$ space, and there is no need to fix a intensive variable (P or T). For univariant equilibria, there are $c + 1$ phases, and either P or T need to be fixed to solve for equilibrium conditions.

Worked example 6.8

Find the independent reactions in the *KFMASH* system considering the assemblage $Grt + Sil + St + WM + Bt + Qz + H_2O$

We will use the delineated approach in Chapter 3 (using the *LU* decomposition)

```
from sympy import symbols, Matrix
import scipy.linalg as LA
from scipy.linalg import lu
import numpy as np
Py, Alm, Sill, Mst, Fst = symbols('Py, Alm, Sill, Mst, Fst')
Mu, Cel, Fcel, Phl = symbols('Mu, Cel, Fcel, Phl')
Ann, East, Obi, Qz, H2O = symbols('Ann, East, Obi, Qz, H2O')
end_members = Matrix([Py, Alm, Sill, Mst, Fst, Mu, Cel, Fcel,
                      Phl, Ann, East, Obi, Qz, H2O])
A = np.array([[3,    1,  0,  3,  0,  0],
              [3,    1,  3,  0,  0,  0],
              [1,    1,  0,  0,  0,  0],
              [7.5,  9,  0,  4,  0,  2],
              [7.5,  9,  4,  0,  0,  2],
              [3,  1.5,  0,  0,0.5,  1],
              [4,  0.5,  0,  1,0.5,  1],
              [4,  0.5,  1,  0,0.5,  1],
              [3,  0.5,  0,  3,0.5,  1],
              [3,  0.5,  3,  0,0.5,  1],
              [2,  1.5,  0,  2,0.5,  1],
              [3,  0.5,  1,  2,0.5,  1],
              [1,    0,  0,  0,  0,  0],
              [0,    0,  0,  0,  0,  1]])
# add columns of zeros
A = np.pad(A, ((0, 0), (0, 8)), mode='constant',
           constant_values=0)
l, u = lu(A, permute_l = True)
C = np.linalg.matrix_rank(A) # components = rank of the matrix
ind_reactions = end_members.rows - C
print("Number of independent reactions: ", ind_reactions)
reactions_array = (LA.inv(l)[C:,:]).round(decimals=15)
reactions = Matrix(reactions_array)*end_members
```

This script will return an array with the independent reactions. Due to rounding errors, it is possible to have some very small non-zero coefficients. Here we ignore values < 1e-15.

Worked example 6.9

Convert reaction coefficients from worked example 6.8 to integers.

For that conversion, we use the following utility function:

```
import numpy as np
from fractions import Fraction
def simplify_coeff(x):
    max = x.max()
    numerators = np.zeros(len(x), dtype=np.int64)
    denominators = np.zeros(len(x), dtype=np.int64)
    for i in range(len(x)):
        f = Fraction(x[i]/max).limit_denominator()
        numerators[i] = f.numerator
        denominators[i] = f.denominator
    lcm_denom = np.lcm.reduce(denominators)
    numerators = (numerators * \
                  lcm_denom / denominators).astype(np.int64)
    gcd_num = np.gcd.reduce(numerators)
    return numerators/gcd_num
```

This function operates by dividing all elements of the array by its maximum (the new maximum will be 1) and then converting float numbers to fractions using the **Fraction()** function from the *fractions* module. Due to the limitations of floating-point arithmetic in computers, we have to use the method **limit_denominator()** to get good results (try to use the above code without the method and see results). The next steps in the function consist of (i) finding the less common multiple of denominators in the fractions to clear all denominators and (ii) find the greatest common divisor of new numerators to convert them to the smallest possible integers.
Calling the function with the first reaction in the array:

```
reaction_1 = np.array(reactions_array[0])
reaction_1s = simplify_coeff(reaction_1)
print(Matrix(reaction_1s).T*end_members)
# Check that reaction is balanced:
print(not np.dot(A.T, reaction_1s).any()) # should return True
```

The complete set of independent reactions is:

```
[4*Alm - 50*Cel - 3*Fst + 56*H2O - 25*Mst + 50*Phl + 248*Sill]
[-4*Cel + 4*Fcel - 1*Fst + 1*Mst]
[-4*Alm + 3*Fst - 3*Mst + 4*Py]
[4*Ann - 3*Fst + 3*Mst - 4*Phl]
[92*Alm + 28*Cel + 124*East - 69*Fst + 48*H2O + 45*Mst - 152*Phl]
[-1*Fst + 1*Mst + 4*Obi - 4*Phl]
[-36*Alm - 46*Cel + 27*Fst - 8*H2O - 23*Mst + 46*Phl + 124*Qz]
[92*Alm - 96*Cel - 69*Fst + 48*H2O + 45*Mst + 124*Mu - 28*Phl]
```

6.5.2.1.1 Class Method to Calculate Independent Reactions. The method within the class that calculates independent reactions with the considered assemblage takes a compositional matrix, makes it square, and then apply *LU* decomposition to find the independent reactions. This function is added in the last block of utility functions. To implement routines for solving for equilibrium conditions using a system of nonlinear equations in the **PhaseEquilibria** class, some additional *Python* modules are needed, as shown below.

```
import scipy.linalg as LA
from scipy.linalg import lu
from fractions import Fraction
...
    def __calcIndependentReactions(self, A, dec = 15):
        (nr, nc) = A.shape
        A_ = np.pad(A, ((0, 0), (0, nr-nc)), mode='constant',
                    constant_values=0)
        l, u = lu(A_, permute_l = True)
        # components = rank of the matrix
        C = np.linalg.matrix_rank(A_)
        reactions_array = (LA.inv(l)[C:,:]).round(decimals=dec)
        R = np.zeros((len(reactions_array), nr))
        for i in range(len(reactions_array)):
            R[i] = self.__simplifyCoefficients(reactions_array[i])
        return R
```

The **__calcIndependentReactions** function needs to be called just before the Gibbs minimization block in the **calculate** function. Note also that there is one more parameter for the **calculate** function (*fixedP*), which is used to decide later which objective function to call in the root-finding algorithm. This new parameter has a default value, so calculations will be made with fixed pressure by default. The class can still be used to do other calculations (e.g., Gibbs minimization) without specifying the new parameter.

```
    ...
    def calculate(self, calc_type, assemb, param, intensive_vars,
                  fixedP = True):
        ...
        self.R = self.__calcIndependentReactions(A_assemb)
        #============================Gibbs minimization block
        if calc_type == "Gibbs_min":
            ...
```

The routine within the **__calcIndependentReactions()** function calls a simplify coefficient function to convert reaction coefficients to integers. This function needs to be included within the class and is added to the utility functions block.

```
    def __simplifyCoefficients(self, x, ld = 1000000):
        max = x.max()
        numerators = np.zeros(len(x), dtype=np.int64)
        denominators = np.zeros(len(x), dtype=np.int64)
        for i in range(len(x)):
            f = Fraction(x[i]/max).limit_denominator(ld)
            numerators[i] = f.numerator
            denominators[i] = f.denominator
        common_denom = np.lcm.reduce(denominators)
        numerators = (numerators * \
                      common_denom / denominators).astype(np.int64)
        common_num = np.gcd.reduce(numerators)
        return numerators/common_num
```

Several objective functions within the class are designed to solve different types of problems (e.g., finding univariants and invariants). If the equilibrium is univariant, then this problem has $c + 1$ phases; for a six component system ($KFMASH$) with seven phases (for example, $Grt + Sil + St + WM + Bt + Qz + H_2O$), there are eight equations (independent reactions) and nine unknowns (a total of eleven endmembers in four solid solutions giving seven compositional variables). To find equilibrium, one variable needs to be fixed. If, for example, pressure is fixed, the objective function must take as arguments (unknowns) the seven compositional variables and temperature. The return value of the objective function is the dot product of the reaction coefficients matrix (independent set of reactions) with the endmembers' chemical potential vector.

```
    #============================== Functions for PTgrid
    def __findGridUnivAtP(self, param):
        self.T = param[-1]
        self.__calcPhaseProperties(param)
        return np.dot(self.R, self.μ)
    def __findGridUnivAtT(self, param):
        pass
    def __findGridInv(self, param):
        pass
    #============================== end of Functions for PTgrid
```

In these objective functions, the **param** array contains the compositional variables and, in the **__findGridUnivAtP**, the temperature (i.e., the variables to be solved for). Note that two more functions are defined here for doing calculations with fixed temperatures and for calculating invariants. Separating routines is an strategy to avoids **if** statements within objective functions, which are going to be called multiple times.

Within the **calculate()** function, an **if** block will route the class method to the root-finding algorithm, that will work with the objective function to calculate the univariant reaction. For the **fixedP** option, this will happen in a range of pressures to solve for phase compositions and temperatures.

```
        #============================PTGrid
        if calc_type == "PTgrid":
            eqIntVars = np.zeros(len(intensive_vars))
            variables = np.zeros((len(intensive_vars),
                                  len(param)-1))
            reactions = []
            if fixedP:
                for i in range(len(intensive_vars)):
                    self.P = intensive_vars[i]
                    opt_result = opt.root(self.__findGridUnivAtP,
                                          param)
                    eqIntVars[i] = opt_result.x[-1]
                    variables[i] = opt_result.x[:-1]
                    comp = self.phases_comp
                    r = self.__calcIndependentReactions(comp,
                                                        dec = 3)
                    reactions.append(r)
            else: # This is for fixed T and invariants
                pass
            self.result = (eqIntVars, variables, reactions)
        #============================PTGrid
```

Note that within the **if** block, there is an instruction to calculate the independent reactions with the equilibrium phase compositions. The calculated compositional variables are also stored, as they are useful to look at variations in phase compositions along the univariant curve (e.g., variation of the $Fe\#$).

Worked example 6.10

Use the modified class to calculate the univariant for the assemblage $Grt + Sil + St + WM + Bt + Qz + H_2O$ in the *KFMASH* system. Plot the reaction in a pressure-temperature diagram and plot the iron number ($Fe\#$) variation in the considered phases along the univariant.

```
univariant = PhaseEquilibria()
pressures = np.arange(3,7.1,0.1)
x_grt = 0.9; x_st = 0.8; x_wm = 0.6; y_wm = 0.8
x_bi = 0.7; y_bi = 0.3; Q_bi = 0.1
x = np.array([x_grt, x_st, x_wm, y_wm, x_bi, y_bi, Q_bi, 650])
univariant.calculate("PTgrid", np.array([0,1,2,3,4,6,7]),
                     x, pressures)
(temp_univ, vars_univ, reactions) = univariant.result
```

Figure 6.5 shows the results of calculations. The variable `reactions` contains the approximated balanced reactions at every point; output integer reaction coefficients are calculated with the method **simplify_coeff** after approximating values to three decimal places.

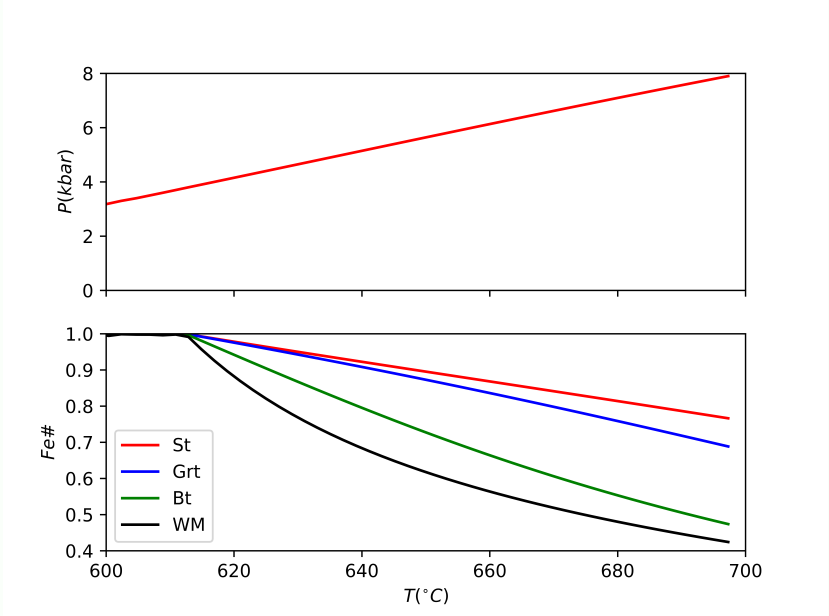

Figure 6.5: Top: univariant *KFMASH* reaction involving $Grt + Sil + St + WM + Bt + Qz + H_2O$. Bottom: Variation in the iron number for phases involved in the univariant reaction.

6.5.2.2. Triangular Compatibility Diagrams

As seen in Chapter 3, triangular compatibility diagrams are generally projections from higher-component systems to three component systems. The procedure, as explained in Chapter 3, consists of the selection of projecting points to reduce the number of components and to bring all phases in the system under consideration onto the triangular projection plane. One important fact to remember is that all phases represented by projecting points must be present in the assemblage when applying these types of diagrams to study rock compositional systems.

Triangular compatibility diagrams are characterized by regions of divariant, trivariant, and quadrivariant equilibria. Regions with a scalene triangle form have divariant equilibria, with corners corresponding to the equilibrium compositions of three phases. Regions bounded by the equilibrium compositions of two phases have trivariant equilibria; if both phases are solid solutions, the region is bounded by two curves, if one is a pure phase, the region is delimited by a curve and a point. In these regions, compatibility lines can be drawn to join the composition of the two phases under equilibrium (see Figure 6.6). There are also small quadrivariant regions representing compositional variations of one solid solution phase.

Divariant equilibria have $c-2$ independent reactions, and by fixing P and T, the procedure delineated above can be used to calculate composition of coexisting phases; trivariant equilibria have $c-3$ independent reactions and, to solve the problem, one more variable needs to be fixed besides P and T.

To add triangular compatibility diagrams calculation functionality to the **PhaseEquilibria** class and compute the compositions of phases and their projection coordinates under equilibrium, the code needs to construct a projection matrix. The **PhaseEquilibria** class is a specialized class for the *KFMASH* system (however, note that the code can be modified to make it more generic for calculations in other systems). With this in mind, the projection matrix is partially hard-coded, with the muscovite center of projection point configured during calculations (the point is the equilibrium composition of muscovite, and it will depend on P and T).

The ***__init__*** method is complemented with a square 6×6 projection matrix. The rows of the matrix correspond to old components. The first three columns correspond to the corners of the projection triangle, while the other three are the centers of projection. The fourth column is initially filled with zeros to be later filled with the equilibrium muscovite composition obtained during calculations:

```
        #                    A   F   M  WM  Qz  H2O
        self.pr = np.array([[0., 0., 0., 0., 1., 0.], # SiO2
                            [1., 0., 0., 0., 0., 0.], # Al2O3
                            [0., 1., 0., 0., 0., 0.], # FeO
                            [0., 0., 1., 0., 0., 0.], # MgO
                            [0., 0., 0., 0., 0., 0.], # K2O
                            [0., 0., 0., 0., 0., 1.]])# H2O
```

The routine must determine whether the calculation is for a divariant assemblage (pressure and temperature fixed) or a trivariant assemblage (with pressure, temperature, and one fixed compositional variable). The **calculate()** function will accept two new parameters: (i) an array with values for the fixed compositional variable, and its index (location) within the list of compositional variables in the `param` array. These new parameters have default values to allow for other types of calculations:

```
    def calculate(self, calc_type, assemb, param,
                  intensive_vars, fixedP = True,
                  fixed_vars=None, fixed_vars_ix=[]):
```

The **if** statement block to calculate AFM triangular diagrams looks like this:

```
        #============================AFM
        if calc_type == "AFM":
            self.P = intensive_vars[0]; self.T = intensive_vars[1]
            if fixed_vars is not None:
                self.indices = fixed_vars_ix
                self.comp_vars = np.zeros((len(fixed_vars),
                                           len(x)+1))
                self.result = []
                for i in range(len(fixed_vars)):
                    var_i = fixed_vars[i]
                    afm_calc = self.__findTriEquilibritium_AFM
                    opt_result = opt.root(afm_calc, param,
                                          args=(var_i))
                    if opt_result.success:
                        idx = self.indices[0]
                        self.comp_vars[i] = np.insert(opt_result.x,
                                                      idx, var_i)
                        afm_proccess = self.__processAFMResult()
                        self.result.append(afm_proccess)
            else:
```

(continues on next page)

(continued from previous page)

```
            afm_calc = self.__findDivEquilibritium_AFM
            opt_result = opt.root(afm_calc, param)
            if opt_result.success:
                self.comp_vars = np.array([opt_result.x])
                self.result = self.__processAFMResult()
    #============================AFM
```

The added block calls three functions: two for finding equilibrium (objective functions) and one for processing the results (calculating the projected triangular coordinates):

```
#============================ Functions for AFM
def __findDivEquilibritium_AFM(self, param):
    self.__calcPhaseProperties(param)
    return np.dot(self.R, self.μ)

def __findTriEquilibritium_AFM(self, param, fixedX):
    param_ = np.insert(param, self.indices[0], fixedX)
    self.__calcPhaseProperties(param_)
    return np.dot(self.R, self.μ)

def __processAFMResult(self):
    # index of WM in assemblage
    ix_WM = np.where(self.assemb==3)[0][0]
    self.pr[:,3] = self.phases_comp[ix_WM]
    pr_i = LA.inv(self.pr)
    result = []
    for i in range(len(self.assemb)):
        if self.assemb[i] not in np.array([3,6,7]):
            # projection from muscovite
            pr_phase_comp = np.dot(pr_i, self.phases_comp[i])
            # add normalized result
            sum = (pr_phase_comp[:3]).sum()
            result.append((pr_phase_comp/sum).round(3))
    return result
#============================ end of Functions for AFM
```

Worked example 6.11

Calculate equilibrium composition for phases and their projection coordinates for the assemblages (i) $Grt+Sil+WM+Bt+Qz+H_2O$ and (ii) $Grt+WM+Bt+Qz+H_2O$ at 5 kbar - 660 °C.

Worked example 6.11 (cont.)

1. Find equilibrium state for assemblages (i). This is a divariant assemblage and the calculation is performed by fixing pressure and temperature. The result is projected triangular coordinates for *Grt*, *Bt*, and *Sil* compositions (Figure 6.6).

```
afm = PhaseEquilibria()
x_grt = 0.9; x_st = 0.8; x_wm = 0.6; y_wm = 0.8
x_bi = 0.7; y_bi = 0.3; Q_bi = 0.1
x = np.array([x_grt, x_wm, y_wm, x_bi, y_bi, Q_bi])
afm.calculate("AFM", np.array([0,1,3,4,6,7]), x, [5, 660])
g_bi_sill_AFM = afm.result
```

2. Find equilibrium state for assemblages (ii). This is a trivariant assemblage and the calculation is performed by fixing pressure, temperature, and a compositional variable. A natural choice is the *Fe#* for biotite. The result is projected triangular coordinates for *Grt* and *Bt* compositions at several values of the fixed compositional variable (Figure 6.6).

```
x = np.array([x_grt, x_wm, y_wm, y_bi, Q_bi])
fe_bi = np.arange(0.85,0.99,0.01) # #x_bi
afm.calculate("AFM", np.array([0,3,4,6,7]), x, [5, 660],
              fixed_vars=fe_bi, fixed_vars_ix=[3])
g_bi_afm = afm.result
```

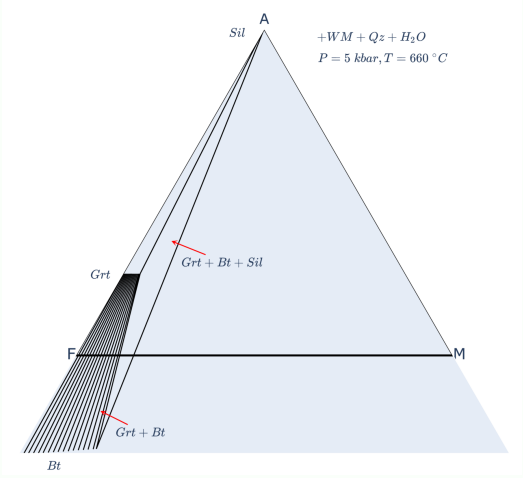

Figure 6.6: Portion of a *AFM* compatibility diagram calculated at 5 kbar - 660 °C. The series of compatibility lines at the bottom left of the triangle represent equilibrium compositions of *Grt* and *Bt*.

6.5.2.3. Pseudosections

The concept of constructing phase diagrams for specific bulk compositions stems from Hensen (1971), who discussed the effect of divariant reactions on phase association stability. In his words, "For a particular bulk composition, the phase assemblages will be bounded partly by portions of the univariant reactions and partly by divariant reactions ..." (Hensen, 1971, p. 197). Later, Labotka (1981) and Powell (1978) used the term *pseudobinary $T - X$ sections*, referring to the characteristics of the diagram carrying information about assemblages but without strict information on composition of phases; however, he involved phases not far from the T-X plane (Powell, 1978). Powell and Downes (1990), Powell and Holland (1990), and Powell *et al.* (1998) popularized the term *pseudosection* to describe diagrams constructed for specific bulk compositions. They used an extended algorithm, present in the program *thermocalc*, for calculating different types of phase diagrams for specific or a range of specific bulk compositions.

Pseudosections are phase diagrams that show only the relevant information for a specific rock composition. Typically, only a few univariant reactions, which bound stable divariant assemblages in a $P - T$ projection or petrogenetic grid, are stable in pseudosections for complex chemical systems. Instead, trivariant and higher variance mineral assemblages are bounded by mode-zero curves, which indicate the appearance and disappearance of a single phase. In pseudosections, the mode of two phases only reaches zero along stable univariant curves and at the intersection of two mode-zero curves. $P - T$ pseudosections are analogous to $P - T$ petrogenetic grids and provide information about changes in mineral assemblages with variations in the intensive variables pressure and temperature. Additionally, pseudosections provide information about pressure-temperature conditions for mineral assemblages and/or mineral compositions in a specific rock composition, modal abundances of phases at equilibrium, proportions of melt coexisting with solid phases, and open-system behavior.

For constructing pseudosections, the algorithm needs mass balance constrain equations for a solvable nonlinear system. There will be a mass balance equation for each component in the system, along with additional variables corresponding to the molar abundance of the phases under equilibrium. The resulting nonlinear system will have n equations with $n + 2$ variables. This means that, for calculating equilibria, two variables need to be fixed. For a $P - T$ pseudosection, for example, curves where one phase reaches zero moles (mode-zero boundaries between fields) can be calculated by setting one molar abundance and either pressure or temperature. By setting

two molar abundances to zero, the algorithm calculates an invariant $P - T$ point (intersection between mode-zero curves).

To add pseudosection calculation functionality to the **PhaseEquilibria** class, a new parameter is needed in the **calculate()** function. This parameter is an array containing phases with zero modal proportions (either one or two phases):

```
    def calculate(self, calc_type, assemb, param,
                  intensive_vars, fixedP = True, fixed_vars=None,
                  fixed_vars_ix=[], zero_modes_ix = []):
```

The **if** block considering pseudosection calculations must include all possible options—that is, calculations for both mode-zero boundaries and their intersections. Currently, only one option is completely coded.

```
        #============================Pseudo
        if calc_type == "PTpseudo":
            self.indices = zero_modes_ix
            eqIntVars = np.zeros(len(intensive_vars))
            moles = np.zeros((len(intensive_vars), len(assemb)))
            variables = np.zeros((len(intensive_vars),
                                  len(param)-len(assemb)))
            if fixedP:
                for i in range(len(intensive_vars)):
                    self.P = intensive_vars[i]
                    func_find_bd = self.__findBoundaryPseudoAtP
                    opt_result = opt.root(func_find_bd, param)
                    if opt_result.success:
                        eqIntVars[i] = opt_result.x[-1]
                        res = opt_result.x
                        moles[i] = np.insert(res[0:len(assemb)-1],
                                             self.indices[0], 0.0)
                        variables[i] = res[len(assemb)-1:-1]
            else: # This is for the other options
                pass
            self.result = (eqIntVars, moles, variables)
        #============================Pseudo
```

The functions that do the actual calculations (called by the pseudosection **if** block coded above) look like this:

```
    #============================ Functions for pseudosection
    def __findIntersectionPseudo(self, param):
        # param contains:
```

(continues on next page)

(continued from previous page)

```
        # variables, moles of phases (except 2 phases), and PT
        pass

    def __findBoundaryPseudoAtT(self, param):
        # param contains:
        # variables, moles of phases (except 1 phase) and P
        # This can be modified to set mode of one phase
        # different than zero
        pass

    def __findBoundaryPseudoAtP(self, param):
        # param contains:
        # variables, moles of phases (except 1 phase) and T
        # This can be modified to set mode of one phase
        # different than zero
        self.T = param[-1]
        moles = np.insert(param[0:(len(self.assemb)-1)],
                          self.indices[0], 0.0)
        vars = param[(len(self.assemb)-1):-1]
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_μ = np.dot(self.R, self.μ)
        return  np.r_[r_mb, r_μ]

    def __findIsopletAtT(self, param):
        # param contains:
        # variables (except 1), moles of phases and P
        pass

    def __findIsopletAtP(self, param):
        # param contains:
        # variables (except 1), moles of phases and T
        pass
    #======================== end of Functions for pseudosection
```

Worked example 6.12

Calculate the curves where staurolite and garnet reaches zero moles (mode zero) in the assemblage $Grt + St + WM + Bt + Qz + H_2O$. Plot the results showing also the univariant reaction calculated in the worked example 6.11.

Worked example 6.12 (cont.)

The requested boundaries are found by calling the function **calculate()** of the *PhaseEquilibria* instantiated *class* with the variable `zero_modes_ix` set to the index of the phase that goes to mode zero. Below is the script to perform these calculations; Figure 6.7 shows a plot with the results.

```
pseudo = PhaseEquilibria(rock_comp = np.array([68.76, 9.87, 7.64,
                                               4.01, 3.16 ,50.]))
pressures = np.arange(7,9.1,0.1)
x_grt = 0.9; x_st = 0.8; x_wm = 0.6; y_wm = 0.8
x_bi = 0.7; y_bi = 0.3; Q_bi = 0.1
x = np.array([1.,1.,1.,1., x_grt,x_st,x_wm,y_wm,
              x_bi,y_bi,Q_bi, 650])
# Staurolite mode zero boundary
pseudo.calculate("PTpseudo", np.array([0,2,3,4,6,7]), x,
                 pressures, zero_modes_ix = [1])
(eqIntVars1, moles, variables) = pseudo.result
# Garnet mode zero boundary
pseudo.calculate("PTpseudo", np.array([0,2,3,4,6,7]), x,
                pressures, zero_modes_ix = [0])
(eqIntVars2, moles, variables) = pseudo.result
```

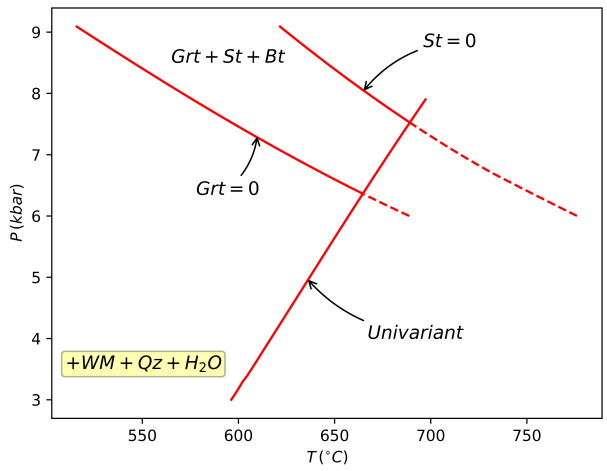

Figure 6.7: Portion of a P-T pseudosection for a specific rock composition showing the curves where staurolite and garnet reach zero moles in the assemblage $Grt+St+WM+Bt+Qz+H_2O$. When crossing the univariant, mode zero curves have a dashed pattern indicating its metastability. The stability field for the assemblage is partially limited by the two mode zero curves and the univariant.

The *PhaseEquilibria class* has not coded methods for calculation of *isopleths* or curves for modal abundances different than zero. However, we can check results from mode-zero calculations to explore variations in modal proportions and phase composition (with compositional variables). The script below extracts information of four points from the garnet mode-zero line, calculated in worked example 6.12 (slicing the output arrays), and then constructs a table using the *Pandas Python* library (Table 6.3).

```
P_slice = pressures[0::100]
T_slice = eqIntVars2[0::100]
m_slice = moles[0::100]
v_slice = variables[0::100]
import pandas as pd
row_names = ['P (kbar)', 'T (žC)', 'Grt', 'St', 'WM', 'Bt', 'Qz',
             r'$H_2O$', r'$X_{Grt}$', r'$X_{St}$', r'$X_{WM}$',
             r'$Y_{WM}$', r'$X_{Bt}$', r'$Y_{Bt}$', r'$Q_{Bt}$']
data = np.round(np.vstack([P_slice[np.newaxis,:],
                           T_slice[np.newaxis,:],
                           m_slice.T, v_slice.T]), 2)
df = pd.DataFrame(data, index=row_names)
```

Table 6.3: Molar proportions (unnormalized) and phase compositions of four points along the garnet mode zero line calculated in worked example 6.12.

P (kbar)	**6**	**7**	**8**	**9**
T (°C)	688.85	625.88	570.96	520.37
Grt	0	0	0	0
St	0.33	0.36	0.39	0.41
WM	2.45	2.64	2.81	2.94
Bt	3.87	3.68	3.51	3.38
Qz	48.61	47.92	47.34	46.87
H_2O	43.02	42.96	42.91	42.87
X_{Grt}	0.85	0.87	0.88	0.9
X_{St}	0.81	0.83	0.85	0.86
X_{WM}	0.55	0.53	0.5	0.47
Y_{WM}	0.92	0.91	0.9	0.88
X_{Bt}	0.64	0.63	0.63	0.63
Y_{Bt}	0.38	0.29	0.21	0.14
Q_{Bt}	0.09	0.1	0.12	0.13

Chapter 7

Thermobarometry

$$\sigma_f^2 = \mathrm{JVJ}^{\mathrm{T}}$$

Estimation of the equilibrium $P-T$ conditions in igneous and metamorphic rocks is an important step in understanding the tectonic evolution of a particular geologic province. The methods applied to obtain meaningful data depend on the type of lithology and equilibrium assemblage observed in the rock. However, the most widely used techniques rely on the knowledge of thermodynamic properties of mineral endmembers and on how they mixed in solid solution phases.

For many phases (but not for all), there is a considerable amount of experimental data to constrain their thermodynamic properties. For many other phases, their thermodynamic data is derived from data of naturally occurring associations and the treatment of both sources of information in internally consistent thermodynamic databases as we mentioned in Chapter 2. In both cases, there is an uncertainty related to the data that propagates to the $P-T$ estimations when these data are used. However, the largest source of uncertainty comes from the mixing models applied to the solid solution. This chapter presents a series of worked examples to understand the methodology for $P-T$ estimations and how errors propagate to the final estimates.

Thermobarometric techniques rely on a set of assumptions that, if unmet, can led to erroneous petrological interpretations. Chemical equilibrium is the first assumption that must be considered. Previous petrographic and microanalytical work allows for checking for disequilibrium textures and consistency of elemental composition between coexisting phases. This is an important first step for selection of appropriate samples and equilibrium volumes within them. For example, laboratory work is necessary for selection of near-rim compositions for estimation of peak $P-T$ conditions when using minerals that show compositional zoning. However, in many cases, it is not possible to find evidence of disequilibrium textures with petrographic observations. Such is the case for disordered phases crystallizing instead of a more stable ordered phase because of kinetics that may not leave any disequilibrium textures evidence (Carpenter & Putnis, 1986).

Another example of disequilibrium is the resetting of phase compositions below peak temperatures without leaving clear evidence of the process. This commonly occurs in granulite facies rocks, and the effect is strongly dependent on elemental diffusion. The second assumption that should be understood is the quality of the underlying thermodynamic data used in $P-T$ calculations. When a researcher is using data for thermobarometry, they rely on a good calibration and/or proper derivation of internally consistent thermodynamic data. However, care must be exercised on the use of published data, considering the restrictions provided by the authors. These

restrictions could come in the form of limitations on the composition of the system and/or the range of pressures and temperatures where the data can be applied.

It is also important to bear in mind the effect of components not included in the system. For example, Mn and Ti have considerable effects on the equilibration of garnet and biotite. This is specially problematic for minor and trace components usually not considered in calculations. Finally, it is also important to understand how the uncertainty in mineral (microprobe) data and activity models propagates to $P-T$ estimations. However, note that uncertainties from experimental work are much larger than those from mineral analyses (Ashworth *et al.*, 2004).

In this chapter, the basic techniques of error propagation are introduced. This is useful for evaluating results in terms of their ability to reproduce independent empirical data, despite the incomplete understanding of all sources of uncertainties, mainly: (i) no good understanding of how endmembers mix in most solid solution phases, and (ii) no good control over the "geologic uncertainty" arising from the sampling and characterization of geological units (Kohn & Spear, 1991; Palin *et al.*, 2016).

7.1. Basic Equations

For a chemical reaction between endmembers of solid solutions, the Gibbs free energy change of the reaction under equilibrium conditions is zero:

$$\triangle_r G_{P,T} = 0 = \triangle_r H^0_{T_0} - T\triangle_r S^0_{T_0} + \int_{T_0}^{T} \triangle_r C_P dT - T\int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT + \int_{P_0}^{P} \triangle_r V\, dP + RT\ln K_{P,T} \tag{7.1}$$

which can be expressed as:

$$0 = \triangle_r G^0 + RT\ln K_{P,T} \tag{7.2}$$

$\triangle_r G^0$ is the standard state Gibbs free energy change of the reaction (pure phases and pressure-temperature of interest). The location of the curve that represents this equation in a pressure-temperature diagram is strongly dependent on the value of $\ln K_{P,T}$ (a continuous reaction) Therefore, this equation is useful for constraining the pressure and temperature of the equilibrium condition. Note that in equation (7.1), the effect of ordering on the

Gibbs free energy of endmembers is ignored. The series of curves, each with a specific equilibrium constant, are called equilibrium constant isopleths, and they have slopes that depend on the thermodynamic parameters of the endmembers participating in the reaction. The partial derivatives of $\ln K_{P,T}$ with respect to P and T provide insight into which parameters control the dependence:

$$\left(\frac{\partial \ln K_{P,T}}{\partial P}\right)_T = -\frac{\partial}{\partial P}\left(\frac{\Delta_r G^0}{RT}\right)_T = -\frac{\Delta_r V}{RT} \tag{7.3}$$

$$\left(\frac{\partial \ln K_{P,T}}{\partial T}\right)_P = -\frac{\partial}{\partial T}\left(\frac{\Delta_r G^0}{RT}\right)_P = \frac{\Delta H_r}{RT^2} \tag{7.4}$$

With the total differential being:

$$d \ln K_{P,T} = \left(\frac{\Delta_r H}{RT^2}\right) dT - \left(\frac{\Delta_r V}{RT}\right) dP \tag{7.5}$$

Note that holding $K_{P,T}$ constant, we obtain:

$$\left(\frac{dP}{dT}\right)_{K_{P,T}} = \frac{1}{T}\frac{\Delta_r H}{\Delta_r V} \tag{7.6}$$

This allows us to conclude that relative differences between $\Delta_r V$ and $\Delta_r H$ determine whether a particular reaction is useful as a geothermometer or as a geobarometer, as these differences control the slope of the reaction curve in $P - T$ space.

As was done in the case of pure phases, the assumption of fixed pressure allows to obtain an equation as a function of temperature only:

$$P = -\frac{\triangle_r H^0_{T_0} + \int_{T_0}^{T} \triangle_r C_P dT}{\triangle_r V} + \frac{\triangle_r S^0_{T_0} + \int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT - R \ln K}{\triangle_r V} * T \tag{7.7}$$

$$\int_{T_0}^{T} \triangle_r C_P dT = T \triangle_r a + \frac{T^2 \triangle_r b}{2} - \frac{\triangle_r c}{T} + 2.0 T^{0.5} \triangle_r d \Bigg|_{T_0}^{T} \tag{7.8}$$

$$\int_{T_0}^{T} \frac{\triangle_r C_P}{T} dT = \triangle_r a \log(T) + T \triangle_r b - \frac{0.5 \triangle_r c}{T^{2.0}} - \frac{2.0 \triangle_r d}{T^{0.5}} \Bigg|_{T_0}^{T} \tag{7.9}$$

Worked example 7.1

Calculate a series of curves representing the endmember reaction for the *GASP* barometer ($2\,Ky + Grs + Qz = 3\,An$) at different constant values of $\ln K_{P,T}$.

Note that this worked example has the purpose of showing the pressure dependence of equation (7.2) as $\ln K_{P,T}$ changes in a geobarometer. For a reaction with plagioclase and garnet non-ideal solid solutions of fixed composition, the value of $\ln K_{P,T}$ might change along the reaction curve, depending on the expressions for the interaction parameters. In this worked example we will use the routines for finding zero roots provided by *SciPy* applied to a *Python* objective function to solve for pressure given temperature and $\ln K_{P,T}$.

```
import numpy as np
from scipy import optimize as opt
def gasp(P,T,lnK):
    R = 0.0083144626
    μo_gr = EM_Gibbs(P,T, dataset["Gr"])
    μo_an = EM_Gibbs(P,T, dataset["An"])
    μo_ky = EM_Gibbs(P,T, dataset["Ky"])
    μo_q = EM_Gibbs(P,T, dataset["Q"])
    residual = μo_gr + 2 * μo_ky + μo_q - 3 * μo_an + \
               R*(T+273.15)*lnK
    return residual
T = np.arange(500., 1200., 20.)
P = np.zeros((5, len(T)))
lnK = np.array([-4,-2,0,2,4])
for i in range(5):
    for j in range(len(T)):
        opt_result = opt.root(gasp, 5, args=(T[j],lnK[i]))
        P[i,j] = opt_result.x[0]
```

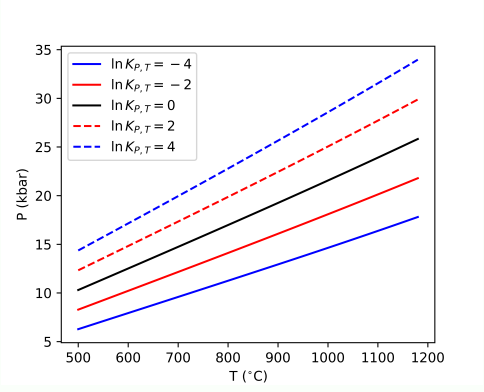

Figure 7.1: P-T diagram of *GASP* reaction curves for different values of $\ln K_{P,T}$

7.2. Phase Equilibria Thermobarometry

7.2.1. Traditional Reaction Thermobarometry

The simplest case for estimating pressure-temperature conditions of equlibrium is that of univariant reactions between pure phases. Some examples of these type of equilibrium are:

- Reactions in the Al_2SiO_5 system: $Kyanite = Andalusite = Sillimanite$
- $Calcite + Quartz = Wollastonite + CO_2$
- $Dolomite = Periclase + Calcite + CO_2$

However, the most commonly used reactions in thermobarometry are those that involve solid solutions. Among these, there are two types of reactions widely used: (i) continuous exchange (thermometer) reactions and (ii) solid-solid net transfer (barometer) reactions.

Continuous exchange reactions involve cation exchange between endmembers. This interchange can be intracrystalline (between different sites in one mineral) or intercrystalline (between sites in two different minerals). Fe^{2+} and Mg^{2+}, being common components of solid solutions and having properties that facilitate substitution, are the most commonly exchanged cations between coexisting silicates. These reactions have relatively high slopes in pressure-temperature diagrams because their ΔS_R is relatively large compared to the ΔV_R. One challenge in applying these types of reactions is the mobility of Fe^{2+} and Mg^{2+}, which means that the (peak temperature) equilibrium is commonly affected by retrogression. These retrograde reactions occur without recrystallization. For the application of these types of equilibria, care must be applied for the selection of zones in minerals without evidences of retrogression and in choosing appropriate refractory phases (e.g., garnet). A second issue in applying these equilibria is the presence of Fe^{3+} in solid solution phases (e.g., biotite and amphibole), as microprobe analysis used routinely for thermobarometry does not discriminate between Fe^{3+} and Fe^{2+}.

Examples of intracrystalline exchange reactions include the partitioning of Fe and Mg between $M1$ and $M2$ sites in orthopyroxenes; this partitioning is known to be greater at low temperatures than at high temperatures. Examples of intercrystalline exchange reactions include the garnet-clinopyroxene ($Pyrope + Hedenbergite = Almandine + Diopside$) and the garnet-biotite thermometers ($Phlogopite + Almandine = Annite + Pyrope$). The latter reaction is known as the *GABI* geothermometer.

Solid-solid net transfer reactions are characterized by a small positive slope due to significantly large molar volume changes in the reaction. Phases commonly involved in these reactions include plagioclase, pyroxene, and garnet. Some widely used barometers include:

- Pyroxene-plagioclase-quartz: $Albite = Jadeite + Quartz$ and $Anorthite = Tschermakite + Quartz$
- *GASP*: $Anorthite = Grossular + Sillimanite + Quartz$
- Garnet-plagioclase-olivine
- Garnet-cordierite-sillimanite-quartz
- Garnet-rutile-ilmenite-sillimanite-quartz

Another type of geothermometers are those that use the composition of coexisting mineral pairs that are structurally related and connected by a miscibility gap or "solvus". This equilibrium is known as solid solution solvus thermometry. In this type of geothermometry, temperature-composition phase diagrams that display the solvus are used together with the composition of the coexisting mineral pair to estimate the temperature of equilibration. Examples of well-calibrated systems include K-feldspar-albite, calcite-dolomite, calcite-siderite, diopside-enstatite, and hedenbergite-ferrosilite.

There is a plethora of calibrated geothermobarometers, the reader is referred to some good compilations in Essene (1982), Bohlen *et al.* (1983), Newton (1983), Bohlen and Lindsley (1987), Essene (1989), Fonarev *et al.* (1991) Holdaway and Mukhopadhyay (1993), Anderson (1996), Anderson *et al.* (2008), Putirka (2008), and Reverdatto *et al.* (2019).

7.2.2. Multiple Reaction Thermobarometry

This technique relies on the application of phase equilibria to $P - T$ estimations using internally consistent thermodynamic databases, giving the opportunity for comparing $P - T$ results from a wide range of mineral reaction thermobarometers. However, be aware that complex solid solution models may break the internal consistency of datasets (Lanari & Duesterhoeft, 2019). The strategies for solving the equilibrium problem in multiple reaction thermobarometry range from weighted least-squares, as implemented in *THERMOCALC* (Powell & Holland, 1988; Powell & Holland, 1994) and in *WEBINVEQ* (Gordon, 1998), to linear programming as implemented in *TWEEQU* (Berman, 1991) and in *WINTWQ* (Berman, 2007). The average $P - T$ routine in *THERMOCALC* uses weighted least

squares to estimate the optimal $P - T$ intersection of an independent, linearized set of reactions. In *WINTWQ*, $P - T$ results are based on specific equilibria (not necessarily an independent set), rather than all possible equilibria, as was done in *TWEEQU*.

7.2.3. Relative Thermobarometry

This term has been applied to the estimation of $P - T$ changes in a rock, as determined by compositional zoning, inclusions, and reaction textures in minerals (Spear, 1989), as well as to thermobarometric estimations in multiple samples with the same mineral assemblage (Worley & Powell, 2000). In both cases, the emphasis of the procedure is on differences of P and T rather than in absolute values. The idea behind both techniques is that, in the context of the problem formulation, uncertainties in solid solution models and thermodynamic data are systematic, meaning that results of (Δ) $P - T$ estimations are more precise.

In the relative thermobarometry application of Spear (1989), the differential form of thermodynamic equations, or the *Gibbs method* (Spear *et al.*, 1982), is used to infer changes in P and T through the understanding of the reaction history of a single sample. The differential forms of the thermodynamic equations allow researchers to monitor changes in the composition of phases under equilibrium as a function of P and T. This is a useful technique for constructing $P - T$ paths.

The relative thermobarometry application of Worley and Powell (2000), as is implemented in *THERMOCALC*, relies on the use of the average $P-T$ routine to estimate $P - T$ differences between samples. This technique can be applied, for example, to gather data for interpreting metamorphic field gradients, which in turn can be used to understand orogenic processes. However, the technique is general in the sense that it can be applied to single-reaction thermometry and barometry (Worley & Powell, 2000).

7.2.4. Phase Diagram Thermobarometry

The use of petrogenetic grids to give qualitative information on $P - T$ conditions of metamorphism dates back to the coining of the term by Bowen (1940), an idea that was further improved by Korzhinskii (1959) with the use of Schreinemakers' analysis. However, the use of this methodology faces the inconvenience of inconsistencies between published petrogenetic grids and, more importantly, the effect of the system composition on the possible potential mineral reactions *seen* in a rock.

Pseudosections are potentially very useful for geothermobarometry since this type of phase diagram allows to make a connection between mineral assemblages seen in the rock with predicted mineral assemblages of pseudosection fields. This comparison has the potential use of retrieving $P-T$ conditions from metamorphic assemblages (Powell & Holland, 2008; Zuluaga *et al.*, 2005). In many cases, pseudosections provide the only available tool for estimating ranges of metamorphic $P-T$ conditions. The technique, however, must address additional challenges besides the ones faced by traditional thermobarometry (uncertainties in thermodynamic data and solid solution models). The two most readily identified are: (i) the determination of the effective bulk composition and (ii) disequilibrium processes, such as reaction overstepping.

These problems have been studied by several authors, and in some cases, possible solutions have been proposed (Evans, 2004; Kelly *et al.*, 2013; Lanari & Duesterhoeft, 2019; Lanari & Engi, 2017; Marmo *et al.*, 2002; Spear, 2017; Spear & Pattison, 2017; Spear & Wolfe, 2018; Stuwe, 1997; Tinkham & Ghent, 2005; Zuluaga *et al.*, 2005). In the case that a good understanding of the rock and the analytical data gives a sense of obtaining a reasonable interpretation from pseudosections, a further step in the interpretation involves assessing conformity between contours for mineral modes and mineral compositions in the phase diagram, and observed values in the rock system. Mineral compositions are routinely obtained from electron probe microanalyzers (*microprobes*), and various techniques exist for obtaining mineral modes in thin sections and sample chips. Mineral modes from thin sections are extrapolated to volumetric modes and these modes need a conversion to molar modes to make the comparison possible.

7.3. Calibration of Traditional Reaction Thermobarometry

There is plenty of literature providing "calibrations" for reactions useful in estimating pressure-temperature equilibrium conditions. Calibrating a geothermometer or a geobarometer involves the application of curve-fitting algorithms to available thermodynamic data using an appropriate equilibrium reaction. Note that a complete consideration of the Gibbs free energy equation for a reaction between solid solutions requires knowledge of the mixing properties of the endmembers in the solid solutions. For simplicity during treatment, some assumptions are made to obtain a simplified Gibbs free energy equation. Common assumptions are that change of

molar volume in the reaction is independent of pressure and temperature and $\Delta C_p = 0$:

$$\Delta_r G_{P,T} = \Delta_r H^0_{1,T} - T\Delta_r S^0_T + P\Delta_r V^0 + RT \ln K_{eq} = 0 \tag{7.10}$$

Rearranging this equation, we get:

$$\Delta_r H^0_{1,T} - T\Delta_r S^0_T = -P\Delta_r V^0 - RT \ln K_{eq} \tag{7.11}$$

The left side of this equation represents the standard state Gibbs free energy change of the reaction (at 1 bar):

$$\Delta_r G^0_{1,T} = \Delta_r H^0_{1,T} - T\Delta_r S^0_T \tag{7.12}$$

Then,

$$\Delta_r G^0_{1,T} = -P\Delta_r V^0 - RT \ln K_{eq} \tag{7.13}$$

The calibration of the geothermometer/geobarometer reaction involves using this equation to determine appropriate parameters in $\Delta_r G^0_{1,T}$ and $\ln K_{eq}$. One approach to do this, for example, is that of Harley (1984) in his derivation of Fe/Mg exchange geothermometer between garnet and orthopyroxene. He used the right-hand side of equation (7.13) to plot experimental data against temperature, and right-hand side of equation (7.12) to estimate the temperature-dependent parameters (y-axis intercept: $\Delta H_{1,T}$ and slope: $-\Delta S^0_T$). His approach used ideal solid solution models to calculate the activities of Fe and Mg endmembers in garnet and orthopyroxene.

A sligthly different approach is used to calibrate the geobarometer garnet-Al_2SiO_5-quartz-plagioclase ($GASP$) in Newton and Haselton (1981) and the clinopyroxene-garnet geobarometer in Beyer *et al.* (2015). In this approach, within the pressure range of interest, a linearized equation of $\Delta G^0_{1,T}$ and the value of ΔV^0 are obtained using data from internally consistent thermodynamic databases and/or by fitting experimental data. The next step involves minimizing the difference between both sides of equation (7.13) using experimental data for refining the less well-constrained activity-composition relationships in the solid solutions. In the barometer calibration of Beyer *et al.* (2015), the linearized $\Delta G^0_{1,T}$ equation and a constant (in the $P - T$ range of interest) suitable value of ΔV^0 were derived from the internally consistent thermodynamic database of Holland and Powell (2011).

In the barometer calibration of Newton and Haselton (1981), they used a linearized standard state Gibbs free energy change equation for the end-member reaction $3\,anorthite \leftrightarrow grossular + 2\,kyanite + quartz$ from Goldsmith (1980). This equation was obtained from high-pressure experiments of anorthite breakdown in the form $P = a + bT$. The calibration proceeds with discussion about appropriate treatment of the terms involving activities and the molar volume change in the reaction. The calibration focuses then on selecting and obtaining the best parameters that produce pressure estimates consistent with experimental results and independently estimated pressures of natural assemblages.

7.3.1. Univariate Robust Regression with Outlier Detection in *Python*

Linear regression can be thought of as finding the straight line that best fits a set of related scattered data points by minimizing the deviations between the points and the line. This line is called the regression line or the best-fit line. The univariate linear regression of n observations with one dependent variable can be formulated as finding the w_0 (intercept) and w (slope) values that best describe the model:

$$y_i = w_0 + w\,x_i + \epsilon_i \tag{7.14}$$

where ϵ_i is the error (residual) associated with observation i. In this model, it is assumed that the ϵ_i values have zero mean and are independently and identically distributed. The model definition is improved by defining a *loss* function to assess how well w_0 and w predict y from x. The most popular loss function is the least squares estimate:

$$Minimize \sum_{i=1}^{n} \epsilon_i^2 \tag{7.15}$$

Thus, in the least squares algorithm, the minimization is performed over the sum of the squares of the differences between observed and estimated values.

Robust regression refers to statistical procedures (estimators) that are insensitive to outliers. Outliers are data points that differs significantly from the other observations; they are usually associated with flawed measurements. Robust regression techniques try to lower the weight of outliers or to remove them during the regression process.

There are several package options in *Python* to perform robust regression, including: *SciPy*, *Scikit-learn*, and *Statsmodels*. We will focus here on the *Scikit-learn* library, which is an open-source *Python* package for data analysis that implements several *machine learning* (*ML*) algorithms. *ML* is a computation technology to train predictive models based on learning from input data. Since linear regression is a fundamental supervised *ML* algorithm, *Scikit-learn* includes linear regression. This *ML* algorithm produces an output (parameters) with the slope and the intercept of a straight line to predict values within a given set of data points.

Scikit-learn includes the *random sample consensus* (*RANSAC*) algorithm (Fischler & Bolles, 1981), which is an iterative regression method which can deal with outliers. The algorithm identifies the inliers and outliers, and the best-fit line is determined only by the identified inliers.

Worked example 7.2

Use equations (7.12) and (7.13) and the *FMAS* experimental run results from Harley (1984) (his Table 1) to derive a geothermometer equation for Fe/Mg exchange geothermometer between garnet and orthopyroxene. The reaction that represents this equilibrium is:

$$\frac{1}{3}Mg_3Al_2Si_3O_{12} + \frac{1}{2}Fe_2Si_2O_6 \leftrightarrow \frac{1}{3}Fe_3Al_2Si_3O_{12} + \frac{1}{2}Mg_2Si_2O_6$$

Use $\Delta V^0 = -22.86\,cal/kbar$ and assume ideal solid solutions and no ordering of Fe and Mg in orthopyroxene:

$$K_{eq} = K_D = \frac{1 - X_{Mg}^{Grt}}{X_{Mg}^{Grt}} \frac{X_{Mg}^{Opx}}{1 - X_{Mg}^{Opx}}$$

---

1. Import data from Harley (1984) into *NumPy* arrays.
2. Calculate $\Delta G_{1,T}^0$ and use a linear regression algorithm from the *Scikit-learn* library to fit the data. *Scikit-learn* has several models; we are interested in one of the robust types, here we use the *RANSAC* regressor. The *RANSAC* model is trained with the data using the **fit()** function, which expects a 2D array, so the 1D input temperatures is converted to 2D (**T[:, np.newaxis]**). Note that the method randomly takes samples, to have reproducible results we set a seed for the random state manually.
3. Calculate the standard errors of the estimated parameters. These errors are the square root of the variances of the estimates. Assuming the model has gaussian error, $\hat{\sigma}^2$ can be calculated from the sample variance of the residuals. There is no calculation of error propagation in the experimentally located endmember reaction. For details on this problem see Kohn and Spear (1991).

Worked example 7.2 (cont.)

```
import numpy as np
from sklearn import linear_model
P = np.array([30, 30, 30, 25, 25, 20, 20, 20, 17.5, 17.5,
              17.5, 15, 15, 15, 12.5, 20, 20, 20, 17.5, 17.5,
              17.5, 15, 15, 15, 12, 12, 12, 10, 10, 10, 10,
              10, 10, 10, 7.5, 7.5, 15, 15, 12.5, 12.5, 12.5,
              10, 10, 10, 7.5, 7.5, 7.5, 5, 5, 5, 10, 10,
              10, 7.5, 7.5, 7.5, 7.5, 7.5, 5, 5, 5, 5, 5, 5])
T = np.array([1473, 1473, 1473, 1473, 1473, 1423, 1423, 1423,
              1423, 1423, 1423, 1423, 1423, 1423, 1423, 1323,
              1323, 1323, 1323, 1323, 1323, 1323, 1323, 1323,
              1323, 1323, 1323, 1323, 1323, 1323, 1323, 1323,
              1323, 1323, 1323, 1323, 1248, 1248, 1248, 1248,
              1248, 1248, 1248, 1248, 1248, 1248, 1248, 1248,
              1248, 1248, 1173, 1173, 1173, 1173, 1173, 1173,
              1173, 1173, 1173, 1173, 1073, 1073, 1073, 1073])
KD = np.array([1.72, 1.63, 1.6, 1.61, 1.05, 1.72, 1.68, 1.59,
               1.64, 1.7, 1.6, 1.63, 1.63, 1.5, 1.52,  1.8,
               1.89,  1.8, 1.84, 1.77, 1.8, 1.7, 1.7, 1.53,
               1.83, 1.99, 1.73, 1.62, 1.65, 1.75, 1.79, 1.79,
               1.72, 1.7, 1.57, 1.55,   2, 1.915, 1.87,  1.8,
               1.81,  1.8, 1.85, 1.84, 1.79, 1.62, 1.66, 1.7,
               1.67,  1.8, 2.12,    2, 2.07, 1.93, 2.05, 2,
               1.98, 2.13, 1.82, 1.81, 2.4, 2.27, 2.15, 2.25])
DVo = -22.86 #* 4.184#J/kbar = -95.64624
R = 8.31446261815324/4.184 # J/molK
DG_1T = -P*DVo - R*T*np.log(KD)
ransac = linear_model.RANSACRegressor(random_state=1)
ransac.fit(T[:, np.newaxis], DG_1T)
DS_T = -ransac.estimator_.coef_[0]   # -1.9499788099374231
DH_1T = ransac.estimator_.intercept_ # -3681.0515717213657
# predicted DG_1T values from experiments
DG_1T_hat = ransac.predict(T[:, np.newaxis])
N = len(T); p = 2
# add a column of ones for the intercept term
T_with_intercept = np.append(np.ones((N,1)), T[:, np.newaxis],
                             axis=1)
residuals = DG_1T - DG_1T_hat
residual_sum_of_squares = residuals.T @ residuals
sigma_squared_hat = residual_sum_of_squares / (N - p)
var_beta_hat = np.linalg.inv(T_with_intercept.T @ \
                             T_with_intercept) * \
                             sigma_squared_hat
H_error = var_beta_hat[0, 0] ** 0.5  # 306.8310405201284
S_error = var_beta_hat[1, 1] ** 0.5  # 0.2361718520685103
```

Worked example 7.2 (cont.)

4. Plot the regression line and experimental data, discriminating between inliers and outliers.

```
_T = np.arange(T.min(), T.max())[:, np.newaxis]
_DG_1T = ransac.predict(_T)
inlier_mask = ransac.inlier_mask_
outlier_mask = np.logical_not(inlier_mask)
fig, ax = plt.subplots()
ax.scatter(T[inlier_mask], DG_1T[inlier_mask],
           color='blue', marker='.', label='Inliers')
ax.scatter(T[outlier_mask], DG_1T[outlier_mask],
           color='red', marker='.', label='Outliers')
ax.plot(_T, _DG_1T, color='royalblue', linewidth=1,
        label='RANSAC regression')
ax.legend(loc='upper left')
ax.set_xlabel("T (K)")
ax.set_ylabel(r'$\Delta G_{1,T}^0 \, (cal)$')
```

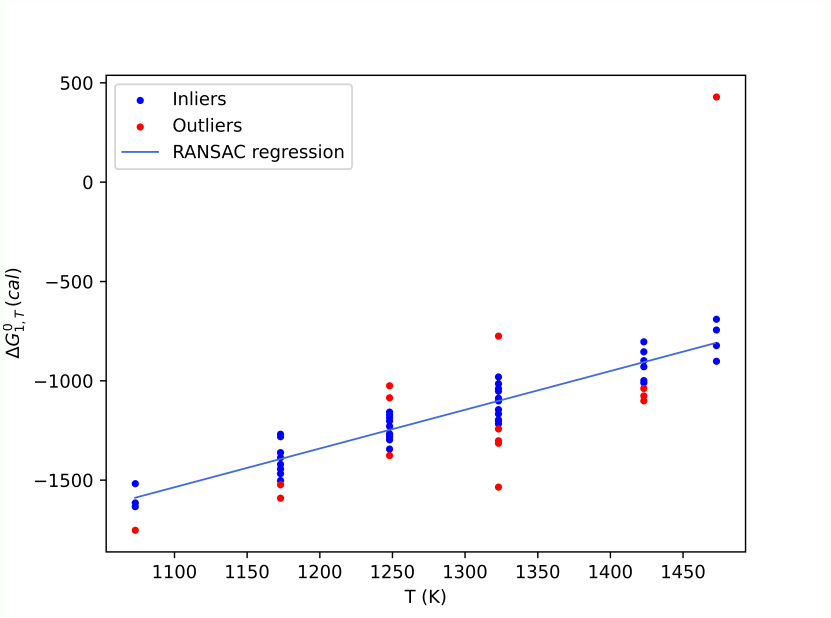

Figure 7.2: Plot of $\Delta G^0_{1,T}$ vs. T for experimental data of a garnetpyroxene barometer in the *FMAS* system from Harley (1984). The line is the result of robust regression on the experimental data points.

Worked example 7.2 (cont.)

5. Write the equation of the calibrated geothermometer.

$$T = \frac{\Delta H^0_{1,T} + P\Delta V^0}{\Delta S^0_T - R \ln K_{eq}}$$

$$T = \frac{3681.05 + 22.86\,P}{1.95 + 1.987 \ln K_{eq}}$$

7.4. Error Propagation in the Gibbs Free Energy Equation

In the last worked example, we calculated the error associated with the regression line based on the sample variance of the residuals. However, there was no formal treatment of the propagation of errors from uncertainties in the input parameters (thermodynamic data and activities). We will address this problem in this section. The general equation for error propagation is:

$$\sigma_f^2 = \mathbf{JVJ}^{\mathrm{T}} \tag{7.16}$$

Where **V** is the covariance matrix:

$$\mathbf{V} = \begin{bmatrix} \sigma^2_{x_1} & \sigma_{x_1x_2} & \sigma_{x_1x_3} & \cdots & \sigma_{x_1x_i} \\ \sigma_{x_2x_1} & \sigma^2_{x_2} & \sigma_{x_2x_3} & \cdots & \sigma_{x_2x_i} \\ \sigma_{x_3x_1} & \sigma_{x_3x_2} & \sigma^2_{x_3} & \cdots & \sigma_{x_3x_i} \\ \vdots & & & \ddots & \vdots \\ \sigma_{x_ix_1} & & & \cdots & \sigma^2_{x_i} \end{bmatrix} \tag{7.17}$$

and **J** is the Jacobian, i.e., a matrix with the first partial differentials of the functions with respect to the variables:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \cdots & \frac{\partial f}{\partial x_i} \end{bmatrix} \tag{7.18}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_i} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_i} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_j}{\partial x_1} & & \cdots & \frac{\partial f_j}{\partial x_i} \end{bmatrix} \tag{7.19}$$

Since $\sigma_{x_i x_j} = \sigma_{x_j x_i}$, the covariance matrix is symmetric. Note that the matrix equation (7.16) for a single function is equivalent to:

$$\sigma_f^2 = \sum_{i=1}^{n} \sigma_{x_i}^2 \left(\frac{\partial f}{\partial x_i}\right)^2 + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \sigma_{x_i x_j} \left(\frac{\partial f}{\partial x_i}\right)\left(\frac{\partial f}{\partial x_j}\right) \tag{7.20}$$

This equation is derived from a Taylor series expansion, ignoring higher-order partial derivatives (Roddick, 1987).

Uncertainties in thermodynamic data for the HP11 dataset are assigned to the enthalpy of formation (ΔH_f^0); thus, the uncertainty of ΔG_R for a particular reaction will be related to uncertainties in ΔH_R and from $\ln K$.

7.4.1. Error Propagation from Uncertainties in ΔH_f^0 to $\Delta_r G$ in Endmember Reactions

The uncertainty in ΔH_R is given by:

$$\sigma_{\Delta H}^2 = \mathbf{J}\mathbf{V}_\mathbf{H}\mathbf{J}^\mathrm{T} \tag{7.21}$$

With the symmetric covariance matrix (lower triangle not shown):

$$\mathbf{V}_{\mathbf{H^0}} = \begin{bmatrix} \sigma_{H_1^0}^2 & \sigma_{H_{12}^0} & \sigma_{H_{13}^0} & \cdots & \sigma_{H_{1i}^0} \\ & \sigma_{H_2^0}^2 & \sigma_{H_{23}^0} & \cdots & \sigma_{H_{2i}^0} \\ & & \sigma_{H_3^0}^2 & \cdots & \sigma_{H_{3i}^0} \\ & & & \ddots & \vdots \\ & & & & \sigma_{H_i^0}^2 \end{bmatrix} \tag{7.22}$$

and the Jacobian:

$$\mathbf{J} = \frac{\partial \Delta_r G}{\partial \Delta \mathbf{H_f^0}} = \frac{\partial \Delta_r H}{\partial \Delta \mathbf{H_f^0}} \tag{7.23}$$

which corresponds to a vector with the coefficients of the balanced endmember reaction.

7.4.2. Error Propagation from Uncertainties in Activities to $\Delta_r G$ in Endmember Reactions

The uncertainties in $\ln K$ come from the uncertainties in the activities. Assuming all endmembers in the reaction belong to different solid solution

phases, the uncertainties in activities (represented by {}) are assumed to be uncorrelated, and the matrix is diagonal:

$$\mathbf{V}_{\{\}} = \begin{bmatrix} \sigma^2_{\{1\}} & & & & \\ & \sigma^2_{\{2\}} & & & \\ & & \sigma^2_{\{3\}} & & \\ & & & \ddots & \\ & & & & \sigma^2_{\{i\}} \end{bmatrix} \tag{7.24}$$

However, remember that some reactions involve more than one endmember from the same solid solution phase, which introduces correlations between uncertainties.

The Jacobian $\mathbf{J}$ is given by the partial differentials of $\ln K$ with respect to the activities of each of the endmembers in the reaction:

$$\mathbf{J} = \frac{\partial \Delta_r G}{\partial \{\mathbf{i}\}} = RT \frac{\partial \ln K}{\partial \{\mathbf{i}\}} \tag{7.25}$$

This is a vector containing the balanced reaction coefficients, each divided by the activity of the corresponding endmember in the partial derivative and multiplied by RT.

7.4.3. Calculation of Uncertainties in Activities

The errors in calculated activities are usually treated as originating from two separate sources: (i) errors coming from the analytical uncertainties of (microprobe) mineral analyses (model-independent) and (ii) errors associated with the activity-composition model. The second source of error is treated as coming from uncertainties in the values of interaction parameters (Ziberna *et al.*, 2017). In Powell and Holland (2008) interaction energy uncertainties are assumed to be ±2 kJ/mol, this value is reduced in Ziberna *et al.* (2017) to ±1 kJ/mol (see also Powell & Holland, 1988).

7.4.3.1. Propagation of Errors from Oxide wt% Analytical Uncertainties to Structural Formula Calculations

We start with the equation for calculating the number of cations per formula unit on an oxygen basis (Giaramita & Day, 1990):

$$afu_i^O = \frac{\left(\frac{w_i}{fw_i}\right) nc_i^{ox} b^O}{S^O} \tag{7.26}$$

$$S^O = \sum_{i=1}^{n} \left(\frac{w_i}{fw_i} \right) no_i^{ox} \tag{7.27}$$

where w_i is the weight percent of the oxide of the *ith* element, nc_i^{ox} is the number of cations in the oxide of the *ith* element, no_i^{ox} is the number of oxygen anions in the oxide of the *ith* element, b^O is the oxygen basis for structural-formula calculation, and fw_i is the formula weight of the oxide of the *ith* element. The partial derivative of afu_i^O with respect to the weight percent of the oxide of the same cation is given by:

$$\frac{\partial afu_i^O}{\partial w_i} = \frac{\left(\frac{nc_i^{ox}}{fw_i} \right) b^O}{(S^O)} - \frac{\left(\frac{w_i nc_i^{ox}}{fw_i} \right) \left(\frac{no_i^{ox}}{fw_i} \right) b^O}{(S^O)^2} \tag{7.28}$$

And for a different cation:

$$\left(\frac{\partial afu_i^O}{\partial w_j} \right)_{j \neq i} = -\frac{\left(\frac{w_i nc_i^{ox}}{fw_i} \right) \left(\frac{no_j^{ox}}{fw_j} \right) b^O}{(S^O)^2} \tag{7.29}$$

For the covariance matrix, we can assume that measured wt% oxides are largely uncorrelated (Kohn & Spear, 1991) and that the uncertainties of analysis are 1% relative (Kohn & Spear, 1991):

$$\mathbf{V} = \begin{bmatrix} (0.01 * w_1)^2 & 0 & 0 & \dots & 0 \\ 0 & (0.01 * w_2)^2 & 0 & \dots & 0 \\ 0 & 0 & (0.01 * w_3)^2 & \dots & 0 \\ \vdots & & & \ddots & \vdots \\ 0 & 0 & 0 & \dots & (0.01 * w_i)^2 \end{bmatrix} \tag{7.30}$$

And the Jacobian:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial afu_1^O}{\partial w_1} & \frac{\partial afu_1^O}{\partial w_2} & \dots & \frac{\partial afu_1^O}{\partial w_i} \\ \frac{\partial afu_2^O}{\partial w_1} & \frac{\partial afu_2^O}{\partial w_2} & \dots & \frac{\partial afu_2^O}{\partial w_i} \\ \vdots & & \ddots & \vdots \\ \frac{\partial afu_i^O}{\partial w_1} & & & \frac{\partial afu_i^O}{\partial w_i} \end{bmatrix} \tag{7.31}$$

The *Python* function to calculate atoms per formula unit (*apfu*) and to propagate errors from oxide wt% analytical uncertainties takes as parameters the phase composition and the number of oxygens in the formula unit and uses vectorized operations to calculate the atoms per formula unit and

the covariance matrix. An utility function (**get_oxide_data()**) is included within the algorithms to retrieve information of oxides, including formula weight, the number of cations, and the number of oxygen atoms.

```
def get_oxide_data(components):
    oxides = {"Al2O3" : [101.96128,2,3], "CaO" : [56.0794,1,1],
              "FeO" : [71.8464,1,1], "H2O" : [18.0152,2,1],
              "K2O" : [94.1954,2,1], "MgO" : [40.3044,1,1],
              "MnO" : [70.9374,1,1], "Na2O" : [61.97894,2,1],
              "TiO2" : [79.866,1,2], "SiO2" : [60.0848,1,2]}
    size = len(components); fw = np.zeros(size)
    nc_ox = np.zeros(size); no_ox = np.zeros(size)
    for i in range(size):
        ox_data = oxides[components[i]]
        fw[i] = ox_data[0]; nc_ox[i] = ox_data[1]
        no_ox[i] = ox_data[2]
    return fw, nc_ox, no_ox
def calc_fu(components, w, bO, error_rel = 0.01):
    size = len(components)
    fw, nc_ox, no_ox = get_oxide_data(components)
    S_O = (w / fw * no_ox).sum()
    afu = w / fw * nc_ox * (bO / S_O)
    V_ox = np.diag((w*error_rel)**2)
    J = np.zeros((size,size))
    for i in range(size):
        J[i,:] = no_ox/fw * w[i] * nc_ox[i] * -bO/(fw[i] * S_O**2)
        J[i,i] += nc_ox[i] * bO / (fw[i] * S_O)
    V_c = np.dot(np.dot(J,V_ox),J.T)
    return (afu, V_c)
```

Worked example 7.3

Use the **calc_fu()** function to calculate *apfu* and covariance matrix for a plagioclase analysis in an eight-oxygen basis (SiO_2 = 58.7, Al_2O_3 = 26.1, CaO = 6.65, Na_2O = 7.73). Evaluate the quality of this plagioclase analysis.

---

1. Calculate *apfu*. Note that although wt% oxides are uncorrelated, the propagated errors in apfu are correlated (for checking this, print the complete covariance matrix).

```
pl_anal = np.array([58.7, 26.1, 6.65, 7.73])
afu, V_c = calc_fu(["SiO2", "Al2O3", "CaO", "Na2O"], pl_anal, 8)
print(afu)                          # [2.636 1.381 0.32  0.673]
print(np.sqrt(np.diag(V_c)))        # [0.011 0.014 0.004 0.008]
```

Worked example 7.3 (Cont.)

2. The (initial) traditional evaluation of the quality of analysis includes a check of oxide weight percent total (accepted values in the range 99 - 100.5 %) and apfu totals (accepted values with 1% relative error).

- Sum of oxides: 99.18
- Sum of cations: 5.01 (octahedral: 0.99, tetrahedral: 4.02)

These results indicate an analysis initially passing the quality check.

3. The next check involves the stoichiometry of the mineral (solid solution model). Assuming no other cations present in the analyzed mineral and a binary solid solution between albite and anorthite, the Al and Si molar proportions of octahedral and tetrahedral sites is fixed by the Ca and Na content of octahedral sites. We will use the plagioclase 2T model introduced previously and check the elemental site distribution using the variable $ca = Ca^A$.

```
SiT_apfu = (afu[0]-2)/2    # 0.3179152461835657
AlT_apfu = 1 - SiT_apfu    # 0.6820847538164343
ca = afu[2]
SiT_stoich = 0.5 - 0.5*ca  # 0.3400323047705345
AlT_stoich = 0.5*ca + 0.5  # 0.6599676952294655
```

The difference amounts to a 3-7% relative error. In the next section, we will work with an approach to adjust the mineral analysis so that it fits the stoichiometry of the solid solution phase.

7.4.3.2. Least Squares Adjustment of Observations - General Equations

The idea of adjustment of mineral analyses using stoichiometry constraints comes from Dollase and Newman (1984), who applied constrained minimization using the method of Lagrange multipliers for finding the nearest stoichiometrically constrained composition to a measured mineral composition. Carson and Powell (1997) suggested approaching the problem using least square adjustment with constraints as presented in Mikhail (1976). The complete set of equations to apply this methodology was presented in Powell and Holland (2008) under the name *ideal analysis approach*. This section presents the general equations of the "least square adjustment" of observations with conditions and constraints as presented in Mikhail (1976) and then the application of the methodology to adjust mineral analysis. Finally, the simplified equations of Powell and Holland (2008) are presented.

The least square adjustment of observations with conditions and constraints provides a mean to obtain a unique set of optimum (stoichiometrical)

estimates for elemental compositions of mineral analysis, using observed oxides weight percent values and an appropriate elemental site distribution model (solid solution model).

The condition equations take the general form:

$$\mathbf{A}(\mathbf{l}+\mathbf{v})+\mathbf{B}\Delta=\mathbf{d} \tag{7.32}$$

or

$$\mathbf{A}\mathbf{v}+\mathbf{B}\Delta=\mathbf{F} \tag{7.33}$$

with $\mathbf{F}=\mathbf{d}-\mathbf{A}\mathbf{l}$ representing the conditions of the model. The functional model can have also constraints (**G**):

$$\mathbf{C}\Delta=\mathbf{G} \tag{7.34}$$

In these equations, $\mathbf{l}$ is the vector of original observations, $\mathbf{v}=\hat{\mathbf{l}}-\mathbf{l}$ is the vector with residuals coming from the difference between observations and a set of estimates that satisfy the model ($\hat{\mathbf{l}}$), and Δ is the vector of corrections. Matrices **A**, **B** and **C** are coefficient matrices coming from the linearization step:

$$\mathbf{A}=\left.\frac{\partial \mathbf{F}}{\partial \mathbf{l}}\right|_{\mathbf{l}^0,\mathbf{x}^0} \tag{7.35}$$

$$\mathbf{B}=\left.\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right|_{\mathbf{l}^0,\mathbf{x}^0} \tag{7.36}$$

$$\mathbf{C}=\left.\frac{\partial \mathbf{G}}{\partial \mathbf{x}}\right|_{\mathbf{x}^0} \tag{7.37}$$

where $\mathbf{x}$ represents the parameters in the model. Note that *condition equation* are any equation that includes one or more observations, while *constraints* equations do not include any observation.

The solution of the above equations, i.e., a set of optimal estimates, is obtained by applying the principle of least squares to minimize the following expression (for details on the derivation, see Mikhail, 1976):

$$\Phi=\mathbf{v}^{\mathrm{T}}\mathbf{W}\mathbf{v} \tag{7.38}$$

here, **W** is the weight matrix of the observations, calculated from the covariance matrix (Σ) and the reference variance (σ_0. If σ_0 is unknown, it is assumed to be equal to one:

$$\mathbf{W}=(\mathbf{Q})^{-1}=\left(\frac{1}{\sigma_0^2}\Sigma\right)^{-1} \tag{7.39}$$

The minimization produces:

$$\mathbf{N} = \mathbf{B}^{\mathbf{t}}\mathbf{W}_{\mathbf{e}}\mathbf{B} = \mathbf{B}^{\mathbf{t}}\left(\mathbf{A}\mathbf{Q}\mathbf{A}^{\mathbf{t}}\right)^{-\mathbf{1}}\mathbf{B} \tag{7.40}$$

$$\mathbf{t} = \mathbf{B}^{\mathbf{t}}\mathbf{W}_{\mathbf{e}}\mathbf{F} = \mathbf{B}^{\mathbf{t}}\left(\mathbf{A}\mathbf{Q}\mathbf{A}^{\mathbf{t}}\right)^{-\mathbf{1}}\mathbf{F} \tag{7.41}$$

$$\mathbf{N}\Delta^{0} = \mathbf{t} \tag{7.42}$$

$$\Delta^{0} = \mathbf{N}^{-\mathbf{1}}\mathbf{t} \tag{7.43}$$

The last equation provides the result of the adjustment applying only the condition equations. In an adjustment with constraints, the solution is given by:

$$\Delta = \Delta^{0} + \mathbf{N}^{-\mathbf{1}}\mathbf{C}^{\mathbf{t}}\mathbf{M}^{-\mathbf{1}}\left(\mathbf{G} - \mathbf{C}\Delta^{0}\right) \tag{7.44}$$

where:

$$\mathbf{M} = \mathbf{C}\mathbf{N}^{-\mathbf{1}}\mathbf{C}^{\mathbf{t}} \tag{7.45}$$

7.4.3.3. Least Squares Adjustment of Observations - Application

In a mineral analyses, the condition equations, with c indicating apfu, are in the form $\hat{c_i} - c_i = 0$, with $\hat{c_i}$ represents estimations of parameters (calculated values from oxides wt%) and c_i the unknown parameters to solve for. Suppose we have a mineral analysis with five observations (five oxides wt%). The number of functional independent parameters is the number of observations minus one; from this, we have four condition equations:

- $c_1 - \hat{c_1} = 0$
- $c_2 - \hat{c_2} = 0$
- $c_3 - \hat{c_3} = 0$
- $c_4 - \hat{c_4} = 0$

There is one more functional condition coming from the model. In general, for mineral analyses, this would be a constraint on the summation of cations, ensuring their sum equals a known value (Σc):

$$c_5 - (\Sigma c - (\hat{c_1} + \hat{c_2} + \hat{c_3} + \hat{c_4})) = 0 \tag{7.46}$$

In the above equations, $\hat{c_i}$ is replaced by (7.26). The matrix with condition equations (**F**) is then:

$$\mathbf{F} = \begin{bmatrix} c_1 - \frac{\left(\frac{w_1}{fw_1}\right)nc_1^{ox}b^O}{S^O} \\ c_2 - \frac{\left(\frac{w_2}{fw_2}\right)nc_2^{ox}b^O}{S^O} \\ c_3 - \frac{\left(\frac{w_3}{fw_3}\right)nc_3^{ox}b^O}{S^O} \\ c_4 - \frac{\left(\frac{w_4}{fw_4}\right)nc_4^{ox}b^O}{S^O} \\ c_5 - \Sigma c + \frac{\left(\frac{w_1}{fw_1}\right)nc_1^{ox}b^O}{S^O} + \frac{\left(\frac{w_2}{fw_2}\right)nc_2^{ox}b^O}{S^O} + \frac{\left(\frac{w_3}{fw_3}\right)nc_3^{ox}b^O}{S^O} + \frac{\left(\frac{w_4}{fw_4}\right)nc_4^{ox}b^O}{S^O} \end{bmatrix} \tag{7.47}$$

Matrix **A** is constructed using the partial derivatives of the condition equations with respect to observations. These elements are calculated using equations (7.28) and (7.29):

$$\mathbf{A} = \begin{bmatrix} \partial f_1/\partial w_1 & \partial f_1/\partial w_2 & \partial f_1/\partial w_3 & \partial f_1/\partial w_4 & \partial f_1/\partial w_5 \\ \partial f_2/\partial w_1 & \partial f_2/\partial w_2 & \partial f_2/\partial w_3 & \partial f_2/\partial w_4 & \partial f_2/\partial w_5 \\ \partial f_3/\partial w_1 & \partial f_3/\partial w_2 & \partial f_3/\partial w_3 & \partial f_3/\partial w_4 & \partial f_3/\partial w_5 \\ \partial f_4/\partial w_1 & \partial f_4/\partial w_2 & \partial f_4/\partial w_3 & \partial f_4/\partial w_4 & \partial f_4/\partial w_5 \\ \partial f_5/\partial w_1 & \partial f_5/\partial w_2 & \partial f_5/\partial w_3 & \partial f_5/\partial w_4 & \partial f_5/\partial w_5 \end{bmatrix} \tag{7.48}$$

Matrix **B** is constructed with partial derivatives of condition equations with respect to parameters (c_i):

$$\mathbf{B} = \begin{bmatrix} \partial f_1/\partial c_1 & \partial f_1/\partial c_2 & \partial f_1/\partial c_3 & \partial f_1/\partial c_4 & \partial f_1/\partial c_5 \\ \partial f_2/\partial c_1 & \partial f_2/\partial c_2 & \partial f_2/\partial c_3 & \partial f_2/\partial c_4 & \partial f_2/\partial c_5 \\ \partial f_3/\partial c_1 & \partial f_3/\partial c_2 & \partial f_3/\partial c_3 & \partial f_3/\partial c_4 & \partial f_3/\partial c_5 \\ \partial f_4/\partial c_1 & \partial f_4/\partial c_2 & \partial f_4/\partial c_3 & \partial f_4/\partial c_4 & \partial f_4/\partial c_5 \\ \partial f_5/\partial c_1 & \partial f_5/\partial c_2 & \partial f_5/\partial c_3 & \partial f_5/\partial c_4 & \partial f_5/\partial c_5 \end{bmatrix} \tag{7.49}$$

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.50}$$

The model can be constrained by any desired combination of equations. The constraint equations must involve parameters but no observations. The number of constraint equations must be less than or equal to the number of parameters. For adjusting mineral analyses, the constraint equations are derived from the model to obtain stoichiometrically adjusted parameters.

One special case is the derivation of Fe^{3+} from microprobe analysis using charge balance equations. To clarify the concept, let us suppose that, for the equations expressed above (a mineral with five oxides wt%), there is only one constrain equation that relates two parameters:

$$-c_2 + (c_3 + 1) = 0 \tag{7.51}$$

Linearization of this equation yields the matrix **C**:

$$\mathbf{C} = [0.0, -1.0, 1.0] \tag{7.52}$$

The function **G** is defined as:

$$\mathbf{G} = c_2 - (c_3 + 1) \tag{7.53}$$

(Note the change in signs between matrix **C** and function **G**).

The *Python* function to adjust mineral analysis using conditions and stoichiometric constraints needs the matrix **A**; this matrix is constructed using the Jacobian from the **calc_fu()** *Python* function. All rows but the last one are equal to the Jacobian multiplied by 1. The last condition equation being a constraint on the total cation sum, Σc; therefore, the last row equals the dot product of a vector of ones with all previous rows of the Jacobian (i.e., the summation of previous rows):

```
    A = -1.0 * J # J is the Jacobian
    for i in range(size):
        # Sc_v is a vector of ones
        A[len(w)-1,i] = np.dot(J[:len(w)-1,i], Sc_v[:-1])
```

Thus, the *Python* function to perform least square adjustment starts by calling the **calc_fu()** function to get initial values of cations and the Jacobian. Next, matrices **A**, **B**, **C**, and **F** are calculated to get the corrections (Δ vector). Additionally, the routine includes calculation of a χ^2 value to do a check on the goodness of fit of the result:

```
def lsq_adj(components, w, b0, Sc, Sc_v = None,
            G = None, error_rel = 0.01):
    afu, Vc, J = calc_fu(components, w, b0, error_rel)
    size = len(w)
    if Sc_v is None: # by default summation of all cations
        Sc_v = np.ones(size)
    A = -1.0 * J
```

(continues on next page)

(continued from previous page)

```
    for i in range(len(w)):
        A[len(w)-1,i] = np.dot(J[:len(w)-1,i], Sc_v[:-1])
    B = np.eye(size)
    f = afu/afu.sum() * Sc
    f[-1] = f[-1] - Sc + np.dot(afu[:-1], Sc_v[:-1])
    f[:-1] = f[:-1] - afu[:-1]

    Q = np.diag((w*error_rel)**2) # assume uncorrelated errors
    Qe = np.dot(A, np.dot(Q,A.T))
    We = np.linalg.inv(Qe)
    N = np.dot(B.T, np.dot(We, B))
    t = np.dot(B.T, np.dot(We, f))
    Ninv = np.linalg.inv(N)
    delta = np.dot(Ninv,t)
    # ====== Adjusted afu - only conditions
    result = afu/afu.sum() * Sc + delta

    if G is not None:
        g_constant = G[-1] # constant
        g = np.dot(G[:-1], result) + g_constant
        C = G[:-1]*-1 # derivatives
        M = np.dot(C, np.dot(Ninv, C.T))
        gCD = g - np.dot(C, delta)
        dD = np.dot(Ninv, np.dot(C.T, 1/M*gCD))
        # ====== Adjusted afu - conditions and constraints
        result = result + dD

    Q_inv = np.linalg.inv(Q)#
    fw, nc_ox, _ = get_oxide_data(components)
    w_adj = result / nc_ox * fw
    w_adj = w_adj / w_adj.sum() * w.sum()
    chi_square = np.dot((w_adj - w).T, np.dot(Q_inv, (w_adj - w)))
    return (result, Vc, chi_square)
```

Note that this *Python* function is written for an adjustment with only one constraint equation. However, it can be easily extended to accept more constraints.

7.4.3.4. Least Squares Adjustment of Observations - A Special Case

The *ideal analysis approach* of Powell and Holland (2008) manipulates equation (7.26) with constraints on the cations to obtain a general constraint

form:

$$\theta^t \mathbf{f_3 p} = \left(\Sigma c\ \mathbf{no^{ox}} - b^O\ \mathbf{nc^{ox}}\ \alpha\right)^t \mathbf{f_3 p} = 0 \tag{7.54}$$

Here, α is a vector representing a linear combination of cations that gives a value of Σc, $\mathbf{f_3}$ is a matrix that allows the calculation of Fe_2O_3 (see Powell & Holland, 2008 for details). $\mathbf{p} = w_i/fw_i$ is a vector with unnormalized mole proportions. The resulting model has only (scaled) observations (**p**), and the solution of the problem is a special case that Mikhail (1976) termed *adjustment of observations only* with no parameters and linear condition equations:

$$\Delta = \mathbf{QA^t W_e F} = \mathbf{QA^t (AQA^t)^{-1} F} \tag{7.55}$$

The coefficient matrix $\mathbf{A} = \theta^t \mathbf{f_3}$ is easily calculated using the general constraint equation. Note, however, equation (7.55) is applied sequentially for each condition equation. Then, for each row in **A**, $\mathbf{A_i Q A_i^t}$ returns a scalar, so:

$$\Delta_i = \frac{\mathbf{QA_i^t F}}{\mathbf{A_i QA_i^t}} \tag{7.56}$$

Worked example 7.4

Use the *Python* **lsq_adj()** function to adjust the plagioclase analysis from worked example 7.3 to a *2T* plagioclase solution model.

In plagioclase, the number of cations in an eight-oxygen basis is five. This is the last condition equation, the others being that the residual in parameter estimations is equals to zero. In a binary solid solution between albite and anorthite, using the *2T* model, the following equations apply: $X^T_{Al} = 0.5 * X_{Ca} + 0.5$ and $X^T_{Si} = 0.5 - 0.5 * X_{Ca}$. Choosing the first equation as the constrain equation and remembering that $Al = 2 * X^T_{Al}$ and $Ca = X_{Ca}$, then $-Al + (Ca + 1) = 0$. From this we get $\mathbf{G} = Al - Ca - 1$ and $\mathbf{C} = [0, -1, 1]$. The arguments passed to the *Python* function are a coefficients array ($O * Si$, $1 * Al$, $-1 * Ca$, $0 * Na$) and the constant (-1) in the **G** function. The last element being the constant (-1): $[0, 1, -1, 0, -1]$.

```
from scipy.stats import chi2
result = lsq_adj(["SiO2", "Al2O3", "CaO", "Na2O"],
                 np.array([58.7, 26.1, 6.65, 7.73]), 8, 5,
                 G=np.array([0,1,-1,0,-1]))
probability = 1 - chi2.cdf(result[2], 4)            #1.41e-05
```

Worked example 7.4 (cont.)

The adjusted parameters are: [2.66, 1.32, 0.32, 0.69]. Results are moderately consistent with the stoichiometry constraint ($X^T_{Al} = 0.5 * X_{Ca} + 0.5$ and $X^T_{Si} = 0.5 - 0.5 * X_{Ca}$); a second iteration will increase the adjustment with the stoichiometry. This adjustment has a large χ^2 and a very small value for χ^2 probability (« 0.05); i.e., the probability that differences between observed and adjusted data are originated by the assumed uncertainties is very small. The large χ^2 could be originated by a model that does not describe the data very well (wrong model and/or bad data) or by underestimation of uncertainties. For example, if we increase the relative error in observed wt% data to 2% relative error, we get a higher χ^2 probability (>10%) and we could say that adjusted data fit observations within this new uncertainty:

```
result = lsq_adj(["SiO2", "Al2O3", "CaO", "Na2O"],
                 np.array([58.7, 26.1, 6.65, 7.73]), 8, 5,
                 G=np.array([0,1,-1,0,-1]), error_rel=0.02)
probability = 1 - chi2.cdf(result[2], 4)            #0.14
```

The *Python* function with the equation for least square adjustment with the *ideal analysis approach* takes as parameters the composition of the mineral phase, the oxygen basis of the mineral unit formula, and the condition equations in the form of α vectors and the corresponding summation of cation values. The function iterates over the condition equations, adding the corrections to the mole proportions calculated in previous steps.

```
def ideal_analysis(components, w, b0, alphas, Sc):
    fw, nc_ox, no_ox = TD_functions.get_oxide_data(components)
    p = w/fw
    Q = np.diag((0.01*p)**2)
    f3 = np.eye(len(w))
    for i in range(len(alphas)):
        theta = Sc[i] * no_ox - b0 * nc_ox * alphas[i]
        Ai = np.dot(theta, f3)
        Fi = np.dot(p,Ai)
        p -= np.dot(Q,np.dot(Ai.T,Fi))/np.dot(Ai,np.dot(Q,Ai.T))
    w_adj = p * fw
    Q_inv = np.linalg.inv(Q)
    chi_square = np.dot((p - w/fw).T, np.dot(Q_inv, (p - w/fw)))
    return (w_adj, chi_square)
```

This function does not calculate Fe_2O_3. To include Fe_2O_3 calculations, you need to modify matrix $\mathbf{f_3}$.

Worked example 7.5

Use the *ideal analysis approach* to adjust the plagioclase composition from worked example 7.3.

For plagioclase with two constraints: (i) $\Sigma c = 5$ (ii) $Al - Ca = 1$

```
w_adj, _ = ideal_analysis(["SiO2", "Al2O3", "CaO", "Na2O"],
                          np.array([58.7, 26.1, 6.65, 7.73]), 8,
                          np.array([[0,1,-1,0],[1,1,1,1]]),
                          np.array([1,5]))
afu, _, _ = calc_fu(["SiO2", "Al2O3", "CaO", "Na2O"], w_adj, 8)
```

The adjusted parameters are: $[2.67, 1.33, 0.32, 0.68]$. Different than in the **lsq_adj** function, the coefficient matrix **A** is constant and there is not need for additional iterations. In this algorithm, the order of the condition equations matters, with the last condition being enforced over the previous conditions (try calling the function with switched rows in *alphas* and the corresponding *Sc* values).

7.4.3.5. Propagation of Errors to Activities

We are now ready to propagate errors from site proportion calculations and interaction parameters to activities. The core of the problem lies in the calculation of the Jacobian:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial\{1\}}{\partial X_1} & \cdots & \frac{\partial\{1\}}{\partial X_j} & \frac{\partial\{1\}}{\partial W_1} & \cdots & \frac{\partial\{1\}}{\partial W_k} \\ \vdots & & \ddots & & & \vdots \\ \frac{\partial\{i\}}{\partial X_1} & \cdots & \frac{\partial\{i\}}{\partial X_j} & \frac{\partial\{i\}}{\partial W_1} & \cdots & \frac{\partial\{i\}}{\partial W_k} \end{bmatrix} \tag{7.57}$$

Where X_j represents the site molar fractions and W_k are the interaction parameters considered in the activity equations ($\{i\}$).

The covariance matrix will have the form:

$$\mathbf{V} = \begin{bmatrix} \sigma^2_{X_1} & \sigma_{X_1X_2} & \cdots & \sigma_{X_1X_j} & 0 & 0 & \cdots & 0 \\ \sigma_{X_1X_2} & \sigma^2_{X_2} & \cdots & \sigma_{X_2X_j} & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & & & & \vdots \\ \sigma_{X_1X_j} & & \cdots & \sigma^2_{X_j} & 0 & 0 & \cdots & 0 \\ 0 & & \cdots & 0 & \sigma^2_{W_1} & 0 & \cdots & 0 \\ \vdots & & & & & & \ddots & \\ 0 & & \cdots & & & 0 & & \sigma^2_{W_k} \end{bmatrix} \tag{7.58}$$

Note that the block matrix formed by the last k rows and columns is a diagonal matrix (as interaction parameters are assumed to have uncorrelated errors).

Worked example 7.6

Calculate the covariance matrix for anorthite and albite activities using the plagioclase analysis of worked example 7.3. Use the following activity equations ({} represents the activity of the endmember in plagioclase): $\{Ab\} = 4 * X_{Na} * X_{Al}^T * X_{Si}^T * \gamma_{ab}$ and $\{An\} = X_{Ca} * (X_{Al}^T)^2 * \gamma_{an}$ with:

$$\gamma_{ab} = e^{\frac{-(1-X_{Na})(-X_{Ca})W_{ab,an}}{RT}}, \quad \gamma_{an} = e^{\frac{-(-X_{Na})(1-X_{Ca})W_{ab,an}}{RT}}$$

---

1. Construct the input covariance matrix. To do this we need to use the matrix resulting from the *lsq_adj* function. However, we need to do one more propagation of error calculation from the calculated apfu to site fractions. From the plagioclase model, we know that $X_{Ca} = Ca$, $X_{Na} = Ca$, $X_{Si}^T = Si/2 - 1$, and $X_{Al}^T = Al/2$. The following code performs the calculation of the new covariance matrix:

```
result, Vc, _ = lsq_adj(["SiO2", "Al2O3", "CaO", "Na2O"],
                        np.array([58.7,26.1,6.65,7.73]),8,5,
                        G = np.array([0,1,-1,0,-1]))
J = np.array([[0.5, 0, 0, 0], # SiT
              [0, 0.5, 0, 0], # AlT
              [0, 0, 1, 0],   # Ca
              [0, 0, 0, 1]])  # Na
Vx = np.dot(J, np.dot(Vc, J.T))
# augmented matrix with sigma2_Wk
V = np.zeros((5,5))
V[:-1,:-1] = Vx
V[4,4] = 1 # sigma2_Wk
from math import exp
Si, Al, Ca, Na = result
AlT, SiT = Al/2, Si/2 - 1
W = 3.1; T = 500 + 273.15; R = 0.0083144626
```

2. Construct the Jacobian matrix for propagation of errors to activities:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial\{Ab\}}{\partial X_{Si}^T} & \frac{\partial\{Ab\}}{\partial X_{Al}^T} & \frac{\partial\{Ab\}}{\partial X_{Ca}} & \frac{\partial\{Ab\}}{\partial X_{Na}} & \frac{\partial\{Ab\}}{\partial W_{ab,an}} \\ \frac{\partial\{An\}}{\partial X_{Si}^T} & \frac{\partial\{An\}}{\partial X_{Al}^T} & \frac{\partial\{An\}}{\partial X_{Ca}} & \frac{\partial\{An\}}{\partial X_{Na}} & \frac{\partial\{An\}}{\partial W_{ab,an}} \end{bmatrix}$$

Worked example 7.6 (cont.)

The partial differentials can be found using *sympy*, for example:

```
import sympy as sp
SiT, AlT, Ca, Na, W, R, T = sp.symbols("SiT, AlT, Ca, Na, W, R, T")
a_ab = 4 * Na * AlT * SiT * sp.exp(((1-Na) * (Ca) * W)/(R*T))
print(sp.diff(a_ab,SiT))
```

Resulting equations are then used to calculate the Jacobian:

```
J11 = 4*AlT*Na*exp(Ca*W*(1 - Na)/(R*T))
J12 = 4*Na*SiT*exp(Ca*W*(1 - Na)/(R*T))
J13 = 4*AlT*Na*SiT*W*(1 - Na)*exp(Ca*W*(1 - Na)/(R*T))/(R*T)
J14 = -4*AlT*Ca*Na*SiT*W*exp(Ca*W*(1 - Na)/(R*T))/(R*T) + \
       4*AlT*SiT*exp(Ca*W*(1 - Na)/(R*T))
J15 = 4*AlT*Ca*Na*SiT*(1 - Na)*exp(Ca*W*(1 - Na)/(R*T))/(R*T)
J22 = 2*AlT*Ca*exp(Na*W*(1 - Ca)/(R*T))
J23 = -AlT**2*Ca*Na*W*exp(Na*W*(1 - Ca)/(R*T))/(R*T) + \
       AlT**2*exp(Na*W*(1 - Ca)/(R*T))
J24 = AlT**2*Ca*W*(1 - Ca)*exp(Na*W*(1 - Ca)/(R*T))/(R*T)
J25 = AlT**2*Ca*Na*(1 - Ca)*exp(Na*W*(1 - Ca)/(R*T))/(R*T)
J = np.array([[J11, J12, J13, J14, J15],  # ab
              [0.0, J22, J23, J24, J25]]) # an
```

Note that the Jacobian is dependent on temperature (we use an arbitrary value of 500 °C).

3. Calculate the covariance matrix for the activities of albite and anorthite in plagioclase:

```
Vact = np.dot(J, np.dot(V, J.T))
std_ab, std_an = np.diag(Vact)**0.5 # 0.0111, 0.0137
```

7.4.4. Putting it all Together

For a schematic reaction: $aA + bB \leftrightarrow cC + dD$, the $\Delta_r H$ and the equilibrium constant are obtained with the equations:

$$\Delta_r H = c\Delta H^0_{f,C} + d\Delta H^0_{f,D} - (a\Delta H^0_{f,A} + b\Delta H^0_{f,B}) \tag{7.59}$$

$$K = \frac{\{C\}^c\{D\}^d}{\{A\}^a\{B\}^b} \tag{7.60}$$

where {} represents the activity of the endmember in its solid solution phase. The uncertainty in $\Delta_r G_{P,T}$ is then:

$$\mathbf{V}_{\Delta_r \mathbf{G_{P,T}}} = \left[\sigma^2_G\right] = \mathbf{J}\mathbf{V}_{\mathbf{\Delta H_f},\{\}}\mathbf{J}^{\mathbf{T}} \tag{7.61}$$

$$\mathbf{V}_{\Delta \mathbf{H}_\mathrm{f},\{\}} = \begin{bmatrix} \mathbf{V}_{\Delta \mathbf{H}_\mathrm{f}} & 0 \\ 0 & \mathbf{V}_{\{\}} \end{bmatrix} \tag{7.62}$$

$$\mathbf{J} = \left[\frac{\partial \Delta_r G_{P,T}}{\partial \Delta \mathbf{H}_\mathrm{f}}, \frac{\partial \Delta_r G_{P,T}}{\partial \{\}} \right] \tag{7.63}$$

$$\mathbf{J} = \left[-a, -b, c, d, -\frac{aRT}{\{A\}}, -\frac{bRT}{\{B\}}, \frac{cRT}{\{C\}}, \frac{dRT}{\{D\}} \right] \tag{7.64}$$

Remember that, usually, $\mathbf{V}_{\{\}}$ is a diagonal matrix (uncorrelated errors in activities).

Worked example 7.7

Use the plagioclase composition from worked example 7.3 and the garnet composition: $SiO_2 = 37.3$, $Al_2O_3 = 21.7$, $FeO = 31.7$, $MgO = 4.32$, $MnO = 0.98$, $CaO = 3.59$ to calculate the uncertainty in $\Delta_r G_{P,T}$ for the GASP barometer reaction at 500 °C and 5 kbar (note that $\Delta_r G_{P,T} \neq 0$ at these conditions).

1. Adjust the garnet composition. We are going to ignore the potential presence of Fe^{3+} in garnet and adjust its analysis using only condition equations (summation of cations equals to 8 in a 12 oxygen basis).

```
result, Vc, _ = lsq_adj(["SiO2","Al2O3","FeO","MgO","MnO","CaO"],
                  np.array([37.3, 21.7,31.7,4.32,0.98,3.59]),12,8)
Si, Al, Fe, Mg, Mn, Ca = result # [2.96,2.04,2.11,0.51,0.07,0.31]
```

2. Compute uncertainties in activities for garnet endmembers at the specified conditions. Activity equations are:

$$\{Py\} = X_{Mg}^3 * \gamma_{Py}$$

$$\begin{aligned} RT \ln \gamma_{Py} = &-(1 - X_{Mg}) * (-X_{Fe}) * W_{Py,Alm} - (1 - X_{Mg}) * (-X_{Mn}) * W_{Py,Spss} \\ &- (1 - X_{Mg}) * (-X_{Ca}) * W_{Py,Gr} - (-X_{Fe}) * (-X_{Mn}) * W_{Alm,Spss} \\ &- (-X_{Fe}) * (-X_{Ca}) * W_{Alm,Gr} - (-X_{Mn}) * (-X_{Ca}) * W_{Gr,Spss} \end{aligned}$$

$$\{Alm\} = X_{Fe}^3 * \gamma_{Alm}$$

$$\begin{aligned} RT \ln \gamma_{Alm} = &-(-X_{Mg}) * (1 - X_{Fe}) * W_{Py,Alm} - (-X_{Mg}) * (-X_{Mn}) * W_{Py,Spss} \\ &- (-X_{Mg}) * (-X_{Ca}) * W_{Py,Gr} - (1 - X_{Fe}) * (-X_{Mn}) * W_{Alm,Spss} \\ &- (1 - X_{Fe}) * (-X_{Ca}) * W_{Alm,Gr} - (-X_{Mn}) * (-X_{Ca}) * W_{Gr,Spss} \end{aligned}$$

$$\{Gr\} = X_{Ca}^3 * \gamma_{Gr}$$

$$\begin{aligned} RT \ln \gamma_{Gr} = &-(-X_{Mg}) * (-X_{Fe}) * W_{Py,Alm} - (-X_{Mg}) * (-X_{Mn}) * W_{Py,Spss} \\ &- (-X_{Mg}) * (1 - X_{Ca}) * W_{Py,Gr} - (-X_{Fe}) * (-X_{Mn}) * W_{Alm,Spss} \\ &- (-X_{Fe}) * (1 - X_{Ca}) * W_{Alm,Gr} - (-X_{Mn}) * (1 - X_{Ca}) * W_{Gr,Spss} \end{aligned}$$

Worked example 7.7 (cont.)

$$\{Spss\} = X_{Mn}^3 * \gamma_{Spss}$$

$$\begin{aligned} RT \ln \gamma_{Spss} = &- (-X_{Mg}) * (-X_{Fe}) * W_{Py,Alm} - (-X_{Mg}) * (1 - X_{Mn}) * W_{Py,Spss} \\ &- (-X_{Mg}) * (-X_{Ca}) * W_{Py,Gr} - (-X_{Fe}) * (-X_{Mn}) * W_{Alm,Spss} \\ &- (-X_{Fe}) * (-X_{Ca}) * W_{Alm,Gr} - (1 - X_{Mn}) * (-X_{Ca}) * W_{Gr,Spss} \end{aligned}$$

We start with the calculation of $\mathbf{V_x}$ from $\mathbf{V_c}$ and construction of an augmented $\mathbf{V}$ matrix with uncertainties for interaction parameters (as in the plagioclase worked example):

```
Vc = Vc[-4:,-4:] # Drop sigmas for Si and Al
Jx = np.array([[1/3, 0, 0, 0], # MgX
               [0, 1/3, 0, 0], # FeX
               [0, 0, 1/3, 0],   # CaX
               [0, 0, 0, 1/3]])  # MnX
Vx = np.dot(Jx, np.dot(Vc, Jx.T))
V = np.zeros((10,10))
V[:-6,:-6] = Vx
V[-6:,-6:] = np.diag(np.ones(6)) # std for W
```

Next, we construct the Jacobian matrix to propagate uncertainties to activities. The derivatives of the activity equations with respect to site fractions and interaction parameters are found using *SymPy*. For example:

```
import sympy as sp
FeX, MgX, MnX, CaX, W, R, T = sp.symbols("FeX, MgX, MnX, CaX,
                                          W, R, T")
Wpyalm, Wpyspss, Wpygr = sp.symbols("Wpyalm, Wpyspss, Wpygr")
Walmspss, Walmgr, Wgrspss = sp.symbols("Walmspss, Walmgr, Wgrspss")
Spy = - (1.0 - MgX ) * (- FeX) * Wpyalm \
      - (1.0 - MgX ) * (- MnX) * Wpyspss \
      - (1.0 - MgX ) * (- CaX) * Wpygr \
      - (- FeX) * (- MnX) * Walmspss  \
      - (- FeX) * (- CaX) * Walmgr \
      - (- MnX) * (- CaX) * Wgrspss
a_py = MgX**3 * sp.exp(Spy/(R*T))
print(sp.diff(a_py,MgX))
```

Then, we use the found derivatives to construct the Jacobian:

$$\mathrm{Jx} = \begin{bmatrix} \frac{\partial\{Py\}}{\partial X_{Mg}} & \frac{\partial\{Py\}}{\partial X_{Fe}} & \frac{\partial\{Py\}}{\partial X_{Ca}} & \frac{\partial\{Py\}}{\partial X_{Mn}} \\ \frac{\partial\{Alm\}}{\partial X_{Mg}} & \frac{\partial\{Alm\}}{\partial X_{Fe}} & \frac{\partial\{Alm\}}{\partial X_{Ca}} & \frac{\partial\{Alm\}}{\partial X_{Mn}} \\ \frac{\partial\{Gr\}}{\partial X_{Mg}} & \frac{\partial\{Gr\}}{\partial X_{Fe}} & \frac{\partial\{Gr\}}{\partial X_{Ca}} & \frac{\partial\{Gr\}}{\partial X_{Mn}} \\ \frac{\partial\{Spss\}}{\partial X_{Mg}} & \frac{\partial\{Spss\}}{\partial X_{Fe}} & \frac{\partial\{Spss\}}{\partial X_{Ca}} & \frac{\partial\{Spss\}}{\partial X_{Mn}} \end{bmatrix}$$

Worked example 7.7 (cont.)

$$Jw = \begin{bmatrix} \frac{\partial\{Py\}}{\partial W_{Py,Alm}} & \frac{\partial\{Py\}}{\partial W_{Py,Gr}} & \frac{\partial\{Py\}}{\partial W_{Py,Spss}} & \frac{\partial\{Py\}}{\partial W_{Alm,Gr}} & \frac{\partial\{Py\}}{\partial W_{Alm,Spss}} & \frac{\partial\{Py\}}{\partial W_{Gr,Spss}} \\ \frac{\partial\{Alm\}}{\partial W_{Py,Alm}} & \frac{\partial\{Alm\}}{\partial W_{Py,Gr}} & \frac{\partial\{Alm\}}{\partial W_{Py,Spss}} & \frac{\partial\{Alm\}}{\partial W_{Alm,Gr}} & \frac{\partial\{Alm\}}{\partial W_{Alm,Spss}} & \frac{\partial\{Alm\}}{\partial W_{Gr,Spss}} \\ \frac{\partial\{Gr\}}{\partial W_{Py,Alm}} & \frac{\partial\{Gr\}}{\partial W_{Py,Gr}} & \frac{\partial\{Gr\}}{\partial W_{Py,Spss}} & \frac{\partial\{Gr\}}{\partial W_{Alm,Gr}} & \frac{\partial\{Gr\}}{\partial W_{Alm,Spss}} & \frac{\partial\{Gr\}}{\partial W_{Gr,Spss}} \\ \frac{\partial\{Spss\}}{\partial W_{Py,Alm}} & \frac{\partial\{Spss\}}{\partial W_{Py,Gr}} & \frac{\partial\{Spss\}}{\partial W_{Py,Spss}} & \frac{\partial\{Spss\}}{\partial W_{Alm,Gr}} & \frac{\partial\{Spss\}}{\partial W_{Alm,Spss}} & \frac{\partial\{Spss\}}{\partial W_{Gr,Spss}} \end{bmatrix}$$

$$J = [Jx\, Jw]$$

Following the same nomenclature for the Jacobian as in the plagioclase example, we define it in *Python*:

```
# First four elements for MgX  FeX  CaX  MnX
# Last element for Ws: pyalm pygr pyspss almgr almspss grspss
               # {Py}
J = np.array([[J11,J12,J13,J14,J15,J16,J17,J18,J19,J110],
               # {Alm}
               [J21,J22,J23,J24, J25,J26,J27,J28,J29,J210],
               # {Gr}
               [J31,J32,J33,J34,J35,J36,J37,J38,J39,J310],
               # {Spss}
               [J41,J42,J43,J44,J45,J46,J47,J48,J49,J410]])
Vact = np.dot(J, np.dot(V, J.T))
std_py, std_alm, std_gr, std_spss = np.diag(Vact)**0.5
# 0.00126 0.00412 0.0004 5.6e-06
```

3. To calculate activities of endmembers in plagioclase and garnet, we need appropriate *SSPhase* objects. The mineral analysis adjustment we made for plagioclase and garnet allow us to use with confidence the solid solution phases modeled with the *SSPhase* class that uses compositional variables (this is not always the case as discussed below). We already have an appropriate plagioclase object, but the garnet object we created in previous chapters is not adequate since only considers mixing between pyrope an almandine. We proceed then by creating an *MnCFMAS* garnet:

```
import sympy as sp
Py, Alm, Gr, Spss = sp.symbols('Py, Alm, Gr, Spss')
FeX, MgX, MnX, CaX = sp.symbols('FeX, MgX, MnX, CaX')
x, c, m = sp.symbols('x, c, m')
prop_eq =  sp.solve([1 - Py - Alm - Gr - Spss,
                  MgX - Py,
                  FeX - Alm,
                  MnX - Spss,
                  CaX - Gr,
                  x - FeX/(FeX+MgX),
                  c - CaX,
                  m - MnX],
                  [Py, Alm, Gr, Spss, FeX, MgX, MnX, CaX])
```

Worked example 7.7 (cont.)

```
vars = [x, c, m]
em = [Py, Alm, Gr, Spss]
sites = [FeX, MgX, CaX, MnX]
act_eq = [MgX**3, FeX**3, CaX**3, MnX**3]
w = [[2.5,0,0], [31,0,0], [2,0,0], [5,0,0], [2,0,0], [0,0,0]]
alphas = [1.0, 1.0, 2.7, 1.0]
dqf = [[], [], [], []]
Grt = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)
```

And calculate activities for endmembers in plagioclase and garnet using the compositional variables for plagioclase ($ca = X_{Ca} = 0.321$) and garnet ($x = X_{Fe}/(X_{Fe} + X_{Mg}) = 0.805$, $c = X_{Ca} = 0.104$, and $m = X_{Mn} = 0.022$):

```
Pl.calc_properties(5, 500, [0.321])
a_ab, a_an = Pl.result[1] # 0.652 0.17
Grt.calc_properties(5, 500, [Fe/(Fe+Mg), Ca/3, Mn/3])
a_py, a_alm, a_gr, a_spss = Grt.result[1] # 0.01 0.339 0.004 0.0
```

4. We now will work on the propagation of errors to $\Delta_r G_{P,T}$ in the *GASP* barometer reaction ($2Ky + Grs + Qz = 3An$). The calculated activities for endmembers were: $\{An\}^{Pl} = 0.17 \pm 0.0137$ and $\{Gr\}^{Grt} = 0.004 \pm 0.0004$ and the covariance matrix $\mathbf{V_{H^0}}$ (extracted from the ds62):

```
#                    ky      gr       qz      an
Vh = np.array([[ 0.4031, 0.2964, -0.0161, 0.3597],
               [ 0.2964, 1.8793,   0.023, 0.8438],
               [-0.0161,  0.023,  0.0654, 0.0187],
               [ 0.3597, 0.8438,  0.0187, 0.5571]])
V = np.zeros((8,8))
V[:4,:4] = Vh
V[5,5] =  std_gr**2 # gr
V[7,7] =  std_an**2 # an
T = 500 + 273.15; R = 0.0083144626
J = np.array([-2, -1, -1, 3, -2*(R*T), -1/a_gr*(R*T),
              -1*(R*T), 3/a_an*(R*T)])
Vg = np.dot(J, np.dot(V, J.T))**0.5 # 1.74
```

7.5. Pressure and Temperature of Intersection of Reactions with Error Estimates

7.5.1. Error Propagation to Estimates of Pressure and Temperature

To calculate pressure at a fixed temperature using equation (7.7), where pressure is expressed as a function of temperature, the Jacobian is given by:

$$\mathbf{J} = \left[\frac{\partial P}{\partial \Delta \mathbf{H_f}}, \frac{\partial P}{\partial \{\}}\right] \tag{7.65}$$

For the schematic reaction $aA + bB \leftrightarrow cC + dD$, the Jacobian for estimating propagated errors to pressure is:

$$\mathbf{J_P} = \left[\frac{a}{\Delta_r V} \; \frac{b}{\Delta_r V} \; \frac{-c}{\Delta_r V} \; \frac{-d}{\Delta_r V} \; \frac{aRT}{\{A\}\Delta_r V} \; \frac{bRT}{\{B\}\Delta_r V} \; \frac{-cRT}{\{C\}\Delta_r V} \; \frac{-dRT}{\{D\}\Delta_r V}\right] \tag{7.66}$$

If instead we are calculating temperature at some fixed pressure, the Jacobian for estimating propagated errors to temperature in the above schematic reaction is:

$$\mathbf{J_T} = \left[-ak_1 \; -bk_1 \; ck_1 \; dk_1 \; \frac{-ak_2}{\{A\}} \; \frac{-bk_2}{\{B\}} \; \frac{ck_2}{\{C\}} \; \frac{dk_2}{\{D\}}\right] \tag{7.67}$$

where:

$$k_1 = \frac{1}{\Delta_r S - R \ln K} \tag{7.68}$$

$$k_2 = \frac{R(\Delta_r H + \Delta_r V P)}{(\Delta_r S - R \ln K)^2} \tag{7.69}$$

Worked example 7.8

Plot the reaction curve for the *GASP* barometer and include the 1σ uncertainty band. Plagioclase: SiO_2 = 58.7, Al_2O_3 = 26.1, CaO = 6.65, Na_2O = 7.73, garnet: SiO_2 = 37.3, Al_2O_3 = 21.7, FeO = 31.7, MgO = 4.32, MnO = 0.98, CaO = 3.59.

Worked example 7.8 (Cont.)

1. Locate the reaction curve using the *optimize* library from *Scipy*

```
from scipy import optimize as opt
import numpy as np
def findGASPAtT(P, T):
    reaction = np.array([-2,-1,-1,3])
    mu_ky = EM_Gibbs(P,T, dataset["Ky"])[0]
    mu_q = EM_Gibbs(P,T, dataset["Q"])[0]
    Grt.calc_properties(P, T, [0.805, 0.104, 0.022])
    mu_gr = Grt.result[2][2]
    Pl.calc_properties(P, T, [0.321])
    mu_an = Pl.result[2][1]
    return np.dot(reaction, [mu_ky, mu_gr, mu_q, mu_an])
T = np.arange(450,800,5); P = np.zeros(len(T))
for i in range(len(T)):
    opt_result = opt.root(findGASPAtT, 5, args=(T[i]))
    P[i] = opt_result.x[0]
```

2. To simplify calculations, we are going to use $\Delta_r V$ at the standard state conditions (1 bar and 25 °C) as extracted from the dataset; this is used to calculate the Jacobian at each point. We also need to calculate activities and errors in activities at each temperature value (see appendix for functions to calculate errors). Vh array is from worked example 7.7.

```
Vo = [dataset["Ky"].V, dataset["Gr"].V,
      dataset["Q"].V, dataset["An"].V]
ΔVr = np.dot(np.array([-2,-1,-1,3]), Vo)
Perror = np.zeros(len(T)); R = 0.0083144626
for i in range(len(T)):
    std_gr = error_grt_act(T[i])[2]
    std_an = error_pl_act(T[i])[1]
    Pl.calc_properties(P[i], T[i], [0.321])
    Grt.calc_properties(P[i], T[i], [0.805, 0.104, 0.022])
    a_gr = Grt.result[1][2]; a_an = Pl.result[1][1]
    V = np.zeros((8,8)); V[:4,:4] = Vh
    V[5,5], V[7,7] = std_gr**2, std_an**2 # gr, an
    J = np.array([-2/ΔVr, -1/ΔVr, -1/ΔVr, 3/ΔVr,
                  -2*(R*(T[i]+273.15))/ΔVr,
                  -1/a_gr*(R*(T[i]+273.15))/ΔVr,
                  -1*(R*(T[i]+273.15))/ΔVr,
                  3/a_an*(R*(T[i]+273.15))/ΔVr])
    Vp = np.dot(J, np.dot(V, J.T))
    Perror[i] = Vp**0.5
```

Worked example 7.8 (Cont.)

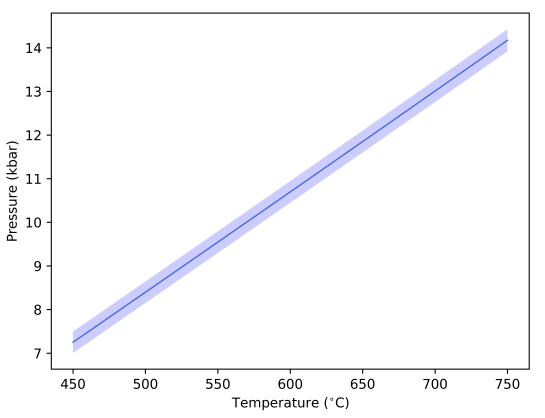

Figure 7.3: $P-T$ location of the GASP barometer reaction with 1σ pressure errors at fixed temperatures.

7.5.2. Error Propagation for Pressure and Temperature Intersection of Reactions

The error propagation equations for these error estimations are:

$$\mathbf{V_{PT}} = \begin{bmatrix} \sigma^2_{P_{int}} & \sigma_{P_{int}T_{int}} \\ \sigma_{P_{int}T_{int}} & \sigma^2_{P_{int}} \end{bmatrix} = \mathbf{J}\mathbf{V}_{\Delta_r \mathbf{H} \ln \mathbf{K}}\mathbf{J}^{\mathrm{T}} \tag{7.70}$$

$$\mathbf{V}_{\Delta_r \mathbf{H} \ln \mathbf{K}} = \begin{bmatrix} \mathbf{V}_{\Delta_r \mathbf{H}} & 0 \\ 0 & \mathbf{V}_{\ln \mathbf{K}} \end{bmatrix} \tag{7.71}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial P_{int}}{\partial \Delta_r \mathbf{H}} & \frac{\partial P_{int}}{\partial \ln \mathbf{K}} \\ \frac{\partial T_{int}}{\partial \Delta_r \mathbf{H}} & \frac{\partial T_{int}}{\partial \ln \mathbf{K}} \end{bmatrix} \tag{7.72}$$

$$\mathbf{J} = \begin{bmatrix} \frac{\partial P_{int}}{\partial \Delta_r \mathbf{H}_{GABI}} & \frac{\partial P_{int}}{\partial \Delta_r \mathbf{H}_{GASP}} & \frac{\partial P_{int}}{\partial \ln \mathbf{K}_{\mathbf{GABI}}} & \frac{\partial P_{int}}{\partial \ln \mathbf{K}_{\mathbf{GASP}}} \\ \frac{\partial T_{int}}{\partial \Delta_r \mathbf{H}_{GABI}} & \frac{\partial T_{int}}{\partial \Delta_r \mathbf{H}_{GASP}} & \frac{\partial T_{int}}{\partial \ln \mathbf{K}_{\mathbf{GABI}}} & \frac{\partial T_{int}}{\partial \ln \mathbf{K}_{\mathbf{GASP}}} \end{bmatrix} \tag{7.73}$$

7.5.2.1. Biotite Model for the *GABI* Thermometer

In a previous worked example, we calculated the reaction curve for the *GASP* barometer. We want to do the same for the *GABI* thermometer. We already have activity data for garnet, next we will calculate activities for biotite in equilibrium with the garnet composition.

For some mineral phases, the formulation of the solution model based on compositional variables may fail to reproduce elemental site distribution as derived directly from a microprobe analysis. In the biotite model constructed in Chapter 5, based on Holland and Powell (2006) and White *et al.* (2014), the octahedral sites are fully occupied (no vacancies), however natural metamorphic biotites have usually a total of octahedral cations between 5.6 and 5.9 and not the expected 6.0 (on a 22 oxygen basis). Most of this deficiency is attributed to vacancies (Guidotti, 1984). Then, we need first to evaluate if we can use the *SSPhase* model constructed and based on compositional variables.

Worked example 7.9

Use least square adjustment to find the ideal analysis for a biotite with composition: SiO_2 = 36.25, TiO_2 = 2.09, Al_2O_3 = 20.35, FeO = 16.88, MgO = 11.51, MnO = 0.02, Na_2O = 0.28, K_2O = 8.56. Use a modified solid solution class to model biotite with the adjusted composition at 500 °C and 5 kbar.

---

1. Let us first convert to cation proportion and evaluate the stoichiometry. For this evaluation we use the equations $Si^T = (Si-2)/2$ and $Al^T = 1-(Si-2)/2$ and compare resulting values with the stoichiometry constraints: $Al^M = Si + Al - 4$, $Si^T = 0.5 - 0.5Al^M$, and $Al^T = 0.5 + 0.5Al^M$:

```
ox = ["Al2O3","SiO2","TiO2","FeO","MgO","MnO","Na2O","K2O"]
anal = np.array([20.35,36.25,2.09,16.88,11.51,0.02,0.28,8.56])
afu,_,_ = calc_fu(oxides, anal, 11)
Si, Ti, Al, Fe, Mg, Mn, Na, K = afu
AlM = Si + Al - 4
print("SiT: ", (Si-2)/2, 0.5 - 0.5*AlM)   # 0.3413 0.2713
print("AlT: ", 1-(Si-2)/2, 0.5 + 0.5*AlM) # 0.6587 0.7287
```

The total occupancy for the octahedral site with a mineral structural formula on a 11-oxygen basis is 2.8895. To do the adjustment in the mineral analysis, we will use the result from the last script for the total summation of cations and add a stoichiometry constraint based on the relation between aluminum in octahedral sites with aluminum in tetrahedral sites: $Al^T = 0.5+0.5Al^M$. With $Al^M = Si+Al-4$ we get: $\sum Al = 2Al^T + Al^M$ and $2 * Si + Al = 7$. The adjusted mineral analyses is:

```
# result = (cat, Vc, chi_2)
result = lsq_adj(ox,anal,11,7.74,G=np.array([1,2,0,0,0,0,0,-7]))
# cation occupancy: [1.81 2.59 0.13 1.07 1.33 0.001 0.04 0.76]
Al,Si,Ti,Fe,Mg,Mn,Na,K = result[0]
AlM = Si + Al - 4
print("SiT: ", (Si-2)/2, 0.5 - 0.5*AlM)   # 0.297 0.300
print("AlT: ", 1-(Si-2)/2, 0.5 + 0.5*AlM) # 0.703 0.700
```

Worked example 7.9 (cont.)

The total occupancy for the octahedral site is now 2.939. Note also that the value of χ^2 is high giving a nearly zero probability (with these data and this model, to obtain a χ^2 probability > 0.05, the relative error has to be increased to 0.03). Additionally, the considered biotite model does not include Ti affecting significantly the Fe and Mg calculated on the octahedral sites. For correcting this last problem in the model, we could construct a new biotite model object with Ti and Mn, but this does not solve the problem of vacancies in octahedral sites. We are going to proceed then by introducing an alternative *SSPhase* class that does not use compositional variables.

2. The modified *SSPhase* class with *apfu* calculations functionality is included in the appendix (*SSPhase_PT*). If needed, order parameters must be added to the vector with *apfu*.

```
import sympy as sp
Phl, Ann, East, Obi = sp.symbols("Phl, Ann, East, Obi")
MgM3, FeM3, AlM3 = sp.symbols("MgM3, FeM3, AlM3")
MgM12, FeM12, SiT, AlT = sp.symbols("MgM12, FeM12, SiT, AlT")
Mg, Fe, Al, Si, Ti, Mn = sp.symbols("Mg, Fe, Al, Si, Ti, Mn")
Q = sp.symbols("Q")
em = [Phl, Ann, East, Obi]
sites = [FeM3, MgM3, AlM3, MgM12, FeM12, SiT, AlT]
act_eq = [4.0 * MgM3 * MgM12**2.0 * SiT * AlT,
          4.0 * FeM3 * FeM12**2.0 * SiT * AlT,
          AlM3 * MgM12**2.0 * AlT**2.0,
          4.0 * FeM3 * MgM12**2.0 * SiT * AlT]
w = [[12,0,0],[10,0,0],[4,0,0],[15,0,0],[8,0,0],[7,0,0]]
alphas = [1,1,1,1]
dqf = [[], [-3,0,0], [], [-3,0,0]]
rx_ordered = [{Obi: -1, Phl: 2/3, Ann: 1/3}]
makes = {Obi: {Phl: 2/3, Ann: 1/3}}
prop_eq = {Phl: MgM3, Ann: FeM12, East: AlM3,
           Obi: FeM3 - FeM12}
site_eq = {FeM3: Fe - 2*(Si+Al+Ti+Fe+Mg+Mn-6)\
                    *(Fe/(Fe+Mg)-Q/3),
           MgM3: Mg - 2*(Si+Al+Ti+Fe+Mg+Mn-6)\
                    *(1-Fe/(Fe+Mg)+Q/3),
           AlM3: Si + Al - 4,
           FeM12: (Si+Al+Ti+Fe+Mg+Mn-6) * (Fe/(Fe+Mg)-Q/3),
           MgM12: (Si+Al+Ti+Fe+Mg+Mn-6) * (1-Fe/(Fe+Mg)+Q/3),
           SiT: (Si-2)/2, AlT: 1-(Si-2)/2}
cations = [Al, Si, Ti, Fe, Mg, Mn, Q]
Bt_PT = SSPhase_PT(em, sites, cations, site_eq, prop_eq,
                   act_eq, w, alphas, dqf, makes, rx_ordered)
comp = np.append(result[0][:-2], 0.01) # Q = 0.01
Bt_PT.calc_properties(10, 500, comp)
```

From this we get all biotite properties at 500 °C and 10 kbar; the resulting order parameter is $Q = 0.1414$.

7.5.2.2. Error Estimation from *Monte Carlo* Simulations

Monte Carlo is a sampling method that uses the mean and the standard deviation of a population to generate a set of random samples, i.e., generates random selections of data from its error probability distribution. The method assumes a certain distribution shape (e.g., normal or lognormal). In *Python*, for example, to generate 10,000 random samples for a population with a mean of -6214.88 and standard deviation of 50 with a normal distribution, we could use:

```
numpy.random.normal(loc=-6214.88, scale=50, size=100000)
```

For error estimation, calculations are repeated with the set of independent variables random samples. The accumulated results define an uncertainty distribution containing the relevant information for the computation of the statistical error. To obtain an accurate uncertainty model, a sufficiently large number of samples need to be generated.

As we have seen in this chapter, error propagation is cumbersome. Additionally, it is not possible to propagate errors for some derived parameters during calculations, such is the case for order parameters resulting from equilibrium with an ordered endmember within a phase. This is where *Monte Carlo* simulations become useful for estimating propagated errors.

7.5.2.3. Intersection of GABI and GASP

To calculate the uncertainties at the intersection of the GASP barometer reaction with the GABI thermometer reaction we need the expressions for $\mathbf{J}$, $\mathbf{V}_{\Delta_r \mathbf{H}}$, and $\mathbf{V}_{\log \mathbf{K}}$ in the (7.71) and (7.72) equations. For obtaining the Jacobian we use the following simplified functions for P_{int} and T_{int} of GABI and GASP:

$$P_{int} = \frac{\Delta_r H_{GABI} \left(R \ln K - \Delta_r S\right)_{GASP} - \Delta_r H_{GASP} \left(R \ln K - \Delta_r S\right)_{GABI}}{-\Delta_r V_{GABI} \left(R \ln K - \Delta_r S\right)_{GASP} + \Delta_r V_{GASP} \left(R \ln K - \Delta_r S\right)_{GABI}} \tag{7.74}$$

$$T_{int} = \frac{\Delta_r H_{GABI} \Delta_r V_{GASP} - \Delta_r H_{GASP} \Delta_r V_{GABI}}{\Delta_r V_{GABI} \left(R \ln K - \Delta_r S\right)_{GASP} - \Delta_r V_{GASP} \left(R \ln K - \Delta_r S\right)_{GABI}} \tag{7.75}$$

Worked example 7.10

Calculate the 1σ standard deviation of the order Q parameter in biotite using *Monte Carlo* simulations. First, generate the set of random samples of standard Gibbs free energy, interaction parameters, and composition for biotite (assume that standard deviations of Gibbs free energy are the same as that of enthalpy of formation). Then, use repeated calculations with the set of random samples and estimate the standard deviation of Q.

```
from numpy.random import Generator, PCG64
rng = Generator(PCG64())
N = 10000
mu_phl = EM_Gibbs(10,500, dataset["Phl"])
mu_ann = EM_Gibbs(10,500, dataset["Ann"])
mu_east = EM_Gibbs(10,500, dataset["East"])
mu_MC = rng.normal(loc=[mu_phl,mu_ann,mu_east],
                   scale=[2.73211, 2.81213, 2.85689],
                   size=[N,3])
wMC = rng.normal(loc=[12.,10.,4.,15.,8.,7.],
                 scale=np.array([1.0,1.0,1.0,1.0,1.0,1.0]),
                 size=[N,6])
comp_loc = np.array([2.59, 1.81, 0.13, 1.07, 1.33, 0.001])
comp_scale = comp_loc*0.01
comp = rng.normal(loc=comp_loc, scale=comp_scale, size=[N,6])

Qs = np.zeros(N)
for i in range(N):
    mus = {"Phl": mu_MC[i,0], "Ann": mu_MC[i,1],
           "East": mu_MC[i,2]}
    ws = np.zeros((6,3))
    ws[:,0] = wMC[i]
    x_bi = np.append(comp[i,:6], 0.01)
    Bt_PT.calc_properties(10,500, x_bi, μo_MC = mus, w_MC = ws)
    Qs[i] = Bt_PT.comp[-1]
print(Qs.std())          # 0.043
```

At 500 °C and 10 kbar, biotite has a value for the order parameter $Q = 0.1414 \pm 0.043$.

With the derivatives of functions in (7.74) and (7.75) with respect to $\Delta_r H$ and $\ln K$

$$k_1 = \frac{\partial P_{int}}{\partial \Delta_r H_{GABI}} \qquad k_2 = \frac{\partial P_{int}}{\partial \Delta_r H_{GASP}} \tag{7.76}$$

$$k_3 = \frac{\partial T_{int}}{\partial \Delta_r H_{GABI}} \qquad k_4 = \frac{\partial T_{int}}{\partial \Delta_r H_{GASP}} \tag{7.77}$$

$$k_5 = \frac{\partial P_{int}}{\partial \ln K_{GABI}} \qquad k_6 = \frac{\partial P_{int}}{\partial \ln K_{GASP}} \tag{7.78}$$

$$k_7 = \frac{\partial T_{int}}{\partial \ln K_{GABI}} \qquad k_8 = \frac{\partial T_{int}}{\partial \ln K_{GASP}} \tag{7.79}$$

the Jacobian becomes:

$$\mathbf{J} = \begin{bmatrix} k_1 & k_2 & k_5 & k_6 \\ k_3 & k_4 & k_7 & k_8 \end{bmatrix} \tag{7.80}$$

By numbering GABI as (1) and GASP as (2) and with

$$ks = R \ln K - \Delta_r S \tag{7.81}$$

The terms in the Jacobian matrix are

$$k_1 = \frac{-ks_2}{-\Delta_r V_{GASP} ks_1 + \Delta_r V_1 ks_2} \tag{7.82}$$

$$k_2 = \frac{ks_1}{-\Delta_r V_{G2} ks_1 + \Delta_r V_1 ks_2} \tag{7.83}$$

$$k_3 = -\frac{\Delta_r V_2}{\Delta_r V_2 ks_1 - \Delta_r V_1 RKS_2} \tag{7.84}$$

$$k_4 = \frac{\Delta_r V_1}{\Delta_r V_2 ks_1 - \Delta_r V_1 ks_2} \tag{7.85}$$

$$k_5 = \frac{R\Delta_r H_2}{-\Delta_r V_2 ks_1 + \Delta_r V_1 ks_2} + \frac{R\Delta_r V_2 \left(\Delta_r H_2 ks_1 - \Delta_r H_1 ks_2\right)}{\left(-\Delta_r V_2 ks_1 + \Delta_r V_1 ks_2\right)^2} \tag{7.86}$$

$$k_6 = -\frac{R\Delta_r H_1}{-\Delta_r V_2 ks_1 + \Delta_r V_1 ks_2} + \frac{R\Delta_r V_1 \left(\Delta_r H_2 ks_1 - \Delta_r H_1 ks_2\right)}{\left(-\Delta_r V_2 ks_1 + \Delta_r V_1 ks_2\right)^2} \tag{7.87}$$

$$k_7 = -\frac{R\Delta_r V_2 \left(\Delta_r H_2 \Delta_r V_1 - \Delta_r H_1 \Delta_r V_2\right)}{\left(\Delta_r V_2 ks_1 - \Delta_r V_1 ks_2\right)^2} \tag{7.88}$$

$$k_8 = \frac{R\Delta_r V_1 \left(\Delta_r H_2 \Delta_r V_1 - \Delta_r H_1 \Delta_r V_2\right)}{\left(\Delta_r V_2 ks_1 - \Delta_r V_1 ks_2\right)^2} \tag{7.89}$$

For obtaining $\mathbf{V}_{\Delta_r \mathrm{H}}$:

$$\mathbf{J}_{\Delta_r \mathrm{H}} = \mathbf{J}_{\Delta_r \mathrm{H}} \mathbf{V}_{\Delta \mathrm{H_f}} \mathbf{J}^T_{\Delta_r \mathrm{H}} \tag{7.90}$$

In the above equation, $\mathbf{V}_{\Delta \mathrm{H_f}}$ is the covariance matrix for ΔH_f^0. The terms in $\mathbf{J}_{\Delta_r \mathrm{H}}$ are given by:

$$\frac{\partial \Delta_r H}{\partial \left(\Delta H_f^0\right)_{em_i}} = (\pm)\, reaction\ coef\ of\ em_i \tag{7.91}$$

With the first row in the matrix representing GABI and the second row representing GASP:

$$\mathbf{J}_{\Delta_r \mathbf{H}} = \begin{bmatrix} -1 & -1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2 & -1 & -1 & 3 \end{bmatrix} \tag{7.92}$$

For obtaining $\mathbf{V}_{\log \mathbf{K}}$:

$$\mathbf{V}_{\log \mathbf{K}} = \mathbf{J}_{\log \mathbf{K}} \mathbf{V}_{\{\}} \mathbf{J}_{\log \mathbf{K}}{}^{T} \tag{7.93}$$

Here, $\mathbf{V}_{\{\}}$ is a diagonal matrix with 1σ standard deviations for activities (no correlation of errors). The terms of $\mathbf{J}_{\log \mathbf{K}}$ are given by:

$$\frac{\partial \log K_{Reaction}}{\partial \{em_i\}} = \frac{(\pm)\, reaction\ coef\ of\ em_i}{\{em_i\}} \tag{7.94}$$

As above, with the first row representing GABI and the second representing GASP:

$$\mathbf{J}_{\log \mathbf{K}} = \begin{bmatrix} \frac{-1}{\{phl\}} & \frac{-1}{\{alm\}} & \frac{1}{\{ann\}} & \frac{1}{\{py\}} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{-2}{\{ky\}} & \frac{-1}{\{grs\}} & \frac{-1}{\{qz\}} & \frac{3}{\{an\}} \end{bmatrix} \tag{7.95}$$

Worked example 7.11

Calculate the location of the *GABI* thermometer in a pressure-temperature diagram using the biotite and garnet compositions from worked examples 7.7 and 7.9; also, calculate the uncertainty on the location of the curve.

---

1. Calculate errors in activities for endmembers in biotite. The propagated error in activities is dependent on temperature as seen in the worked examples for plagioclase and garnet. For simplicity, here we are going to calculate the error at 500 °C and 10 kbar using *Monte Carlo* simulations and assume that error for other conditions.

```
from numpy.random import Generator, PCG64
rng = Generator(PCG64()); N = 10000
mu_phl = EM_Gibbs(10,500, dataset["Phl"])
mu_ann = EM_Gibbs(10,500, dataset["Ann"])
mu_east = EM_Gibbs(10,500, dataset["East"])
mu_MC = rng.normal(loc=[mu_phl,mu_ann,mu_east],
                   scale=[2.73211, 2.81213, 2.85689], size=[N,3])
```

Worked example 7.11 (cont.)

```
wMC = rng.normal(loc=[12.,10.,4.,15.,8.,7.],
                 scale=np.ones(6), size=[N,6])
comp_loc = np.array([2.59, 1.81, 0.13, 1.07, # Si, Al, Ti, Fe
                     1.33, 0.001, 0.1414])   # Mg, Mn, Q
comp_scale = comp_loc*0.01; comp_scale[-1] = 0.043
comp = rng.normal(loc=comp_loc, scale=comp_scale, size=[N,7])
activities = np.zeros((N,4))
for i in range(N):
    mus = {"Phl": mu_MC[i,0], "Ann": mu_MC[i,1],
           "East": mu_MC[i,2]}
    ws = np.zeros((6,3)); ws[:,0] = wMC[i]
    x_bi = comp[i,:]
    Bt_PT.calc_properties(10,500, x_bi, calc_order=False,
                          μo_MC = mus, w_MC = ws)
    activities[i] = Bt_PT.result[1]
print(activities.std(0)) # [0.0131 0.0045 0.0117 0.0239]
```

2. Find the reaction curve and the uncertainty band.

```
from scipy import optimize as opt
X_bt = np.array([2.59,1.81,0.13,1.07,1.33,0.001,0.01])
def findGABIAtP(T, P): # phl+alm=ann+py
    reaction = np.array([-1,-1,1,1])
    Grt.calc_properties(P, T, [0.805, 0.104, 0.022])
    mu_py, mu_alm = Grt.result[2][:2]
    Bt_PT.calc_properties(P, T, X_bt)
    mu_phl, mu_ann = Bt_PT.result[2][:2]
    return np.dot(reaction, [mu_phl, mu_alm, mu_ann, mu_py])
P = np.arange(1,20,0.5); T = np.zeros(len(P))
for i in range(len(P)):
    opt_result = opt.root(findGABIAtP, 500, args=(P[i]))
    T[i] = opt_result.x[0]
Vo = [dataset["Phl"].V, dataset["Alm"].V,
      dataset["Ann"].V, dataset["Py"].V]
reaction = np.array([-1,-1,1,1])
ΔVr = np.dot(reaction, Vo)
#                phl       alm      ann       py
Vh=np.array([[ 7.4644, -0.1195,  6.8048,   0.456],
             [-0.1195,   1.459,  0.7224,   0.404],
             [ 6.8048,  0.7224,  7.9081, -0.1277],
             [  0.456,   0.404, -0.1277,   0.997]])
Terror = np.zeros(len(P))
std_phl, std_ann = 0.0131, 0.0045
```

Worked example 7.11 (cont.)

```
for i in range(len(P)):
    std_py, std_alm = TD_functions.error_grt_act(T[i])[:2]
    Grt.calc_properties(P, T, [0.805, 0.104, 0.022])
    a_py, a_alm = Grt.result[1][:2]
    Bt_PT.calc_properties(P[i], T[i], X_bt)
    a_phl, a_ann = Bt_PT.result[1][:2]
    V = np.zeros((8,8))
    V[:4,:4] = Vh
    V[4:,4:] =  np.diag(np.array([std_phl**2, std_alm**2,
                                  std_ann**2, std_py**2]))
    R = 0.0083144626
    J = np.array([-1/ΔVr, -1/ΔVr, 1/ΔVr, 1/ΔVr,
                  -1*(R*(T[i]+273.15))/ΔVr,
                  -1/a_gr*(R*(T[i]+273.15))/ΔVr,
                  1*(R*(T[i]+273.15))/ΔVr,
                  1/a_an*(R*(T[i]+273.15))/ΔVr])
    Vt = np.dot(J, np.dot(V, J.T))
    Terror[i] = Vt**0.5
```

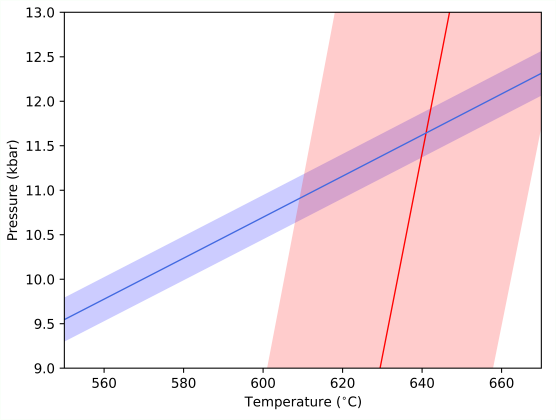

Figure 7.4: Location of *GABI* and *GASP* reactions with 1σ errors.

7.5.2.4. The Uncertainty Ellipse

The uncertainty ellipse, or covariance error ellipse, represents a confidence region for normally distributed data. The inclination of the ellipse is determined by the covariance between variables. If the covariance is zero, the ellipse is aligned with the X and Y axes. For general ellipses, the eigenvectors of the covariance matrix represent the directions in which there is a

larger expected variation of data (*principal components*), i.e., the minor and major axes of the ellipse. The eigenvalues are the variance of the data in the direction of the eigenvectors, i.e., the semi-width of the ellipse's axes.

Error ellipses can be drawn for different confidence intervals that will determine the relative size of the ellipse. The size of the ellipse is obtained by calculating the probability that the sum of squares of independent, normally distributed data samples equals a specific value (n^2) using the cumulative chi-square distribution. For example, with one degree of freedom: $n^2 = 4.0$ for 95.45% (2σ) probability that the data will fall inside the ellipse, and $n^2 = 1.0$ for 68.3% (1σ) probability that the data will fall inside the ellipse.

For an axis-aligned ellipse (uncorrelated errors), the equation for the error ellipse is:

$$\left(\frac{X}{\sigma_X}\right)^2 + \left(\frac{Y}{\sigma_Y}\right)^2 = n^2 \tag{7.96}$$

For general ellipses (correlated errors), a new coordinate system is defined using the eigenvectors. The ellipse is drawn in this new coordinate system, with the eigenvalues representing the variance of the data along the direction of the eigenvectors:

$$\left(\frac{X'}{\lambda_{X'}}\right)^2 + \left(\frac{Y'}{\lambda_{Y'}}\right)^2 = n^2 \tag{7.97}$$

To obtain the ellipse in the original coordinate system, it is rotated by an angle calculated from the orientation of the eigenvectors with respect to the original coordinate system:

$$\theta = \tan^{-1}\frac{\mathbf{V}_1(y)}{\mathbf{V}_1(x)} \tag{7.98}$$

where $\mathbf{V}_1$ is the eigenvector corresponding to the largest eigenvalue.

The *Python* code to construct the error ellipse and the rectangle circumscribing the ellipse, calculated with the covariance matrix, is shown below (Figure 7.5). The width and length of the rectangle circumscribing the ellipse depend on n and the uncertainties on P and T of the intersection.

```
from scipy.stats import chi2
from matplotlib import patches
def error_ellipse(T_int, P_int, cov, confidence=0.683):
    sigmaP = np.sqrt(cov[0,0])
```

(continues on next page)

(continued from previous page)

```
    sigmaT = np.sqrt(cov[1,1])
    n2 = chi2.isf(1-confidence, 1)
    w, v = np.linalg.eigh(cov)
    order = w.argsort()[::-1]
    w, v = w[order], v[:,order]
    theta = np.degrees(np.arctan2(*v[:,0]))
    a=2.*np.sqrt(n2*w[0])
    b=2.*np.sqrt(n2*w[1])
    ellipse = patches.Ellipse(xy=(T_int, P_int),
                              width=a, height=b, angle=theta,
                              facecolor='none', edgecolor='k')
    rectangle = patches.Rectangle((T_int - sigmaT*n2**0.5,
                                   P_int - sigmaP*n2**0.5),
                                   2*sigmaT*n2**0.5,
                                   2*sigmaP*n2**0.5,
                                  facecolor='none', edgecolor='k')
    return (ellipse, rectangle)
```

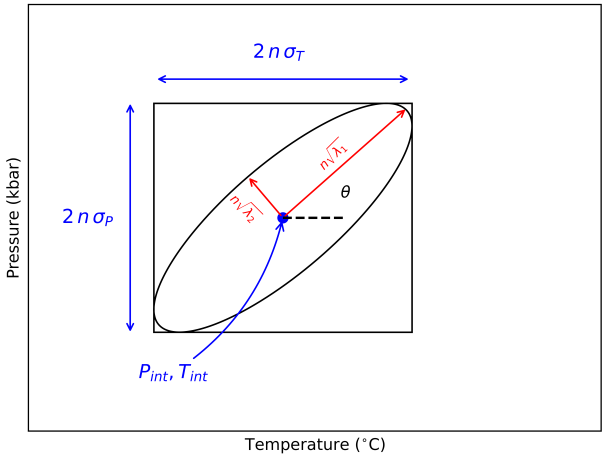

Figure 7.5: Uncertainty ellipse for $P - T$ estimation of mineral reactions intersection. See text for explanation.

Worked example 7.12

Find the intersection of the *GABI* and *GASP* reactions from worked examples 7.8 and 7.11 and plot the 68.3% confidence uncertainty ellipse.

---

1. To find the intersection we use a root-finding algorithm from *Scipy*:

```
from scipy import optimize as opt
def find_int_GABI_GASP(pt):
    T, P = pt
    reaction = np.array([-2,-1,-1,3]) # 2Ky+Grs+Qz=3An
    mu_ky = EM_Gibbs(P,T, dataset["Ky"])
    mu_q = EM_Gibbs(P,T, dataset["Q"])
    Grt.calc_properties(P, T, [0.805, 0.104, 0.022])
    mu_py, mu_alm, mu_gr = Grt.result[2][:3]
    Pl.calc_properties(P, T, [0.332])
    mu_an = Pl.result[2][1]
    r1 = np.dot(reaction, [mu_ky, mu_gr, mu_q, mu_an])
    reaction = np.array([-1,-1,1,1]) # phl+alm=ann+py
    Bt_PT.calc_properties(P, T, X_bt)
    mu_phl, mu_ann = Bt_PT.result[2][:2]
    r2 = np.dot(reaction, [mu_phl, mu_alm, mu_ann, mu_py])
    return np.array([r1,r2])
opt_result = opt.root(find_int_GABI_GASP, np.array([700,16]))
Tint, Pint = opt_result.x  # 641 11.6
```

2. Find the uncertainty in pressure and temperature estimations with equations (7.71), (7.72), and (7.82) to (7.95), and calculated activities and uncertainties for activities in worked examples 7.6, 7.7, 7.10, and 7.12:

```
from math import log
R = 0.0083144626
std_phl, std_ann, std_an = 0.0131, 0.0045, 0.0114
std_alm, std_py, std_gr = 0.004, 0.00093, 3.31e-06
Vact=np.diag(np.array([std_phl, std_alm, std_ann,
                       std_py, 0, std_gr, 0, std_an])**2)
a_phl, a_ann, a_an = 0.1170, 0.0612, 0.1651
a_py, a_alm, a_grs = 0.0087, 0.3408, 0.0031
JlnK_gabi = np.array([-1/a_phl, -1/a_alm, 1/a_ann, 1/a_py])
JlnK_gasp = np.array([-2/1, -1/a_grs, -1/1, 3/a_an])
JlnK2 = np.block([[JlnK_gabi, np.zeros(4)],
                  [np.zeros(4), JlnK_gasp]])
```

Worked example 7.12 (cont.)

```
VlnK = np.dot(np.dot(JlnK,Vact),JlnK.T)
#                        phl      alm      ann       py
Vh_gabi=np.matrix([[ 7.4644, -0.1195,  6.8048,   0.456],
                   [-0.1195,   1.459,  0.7224,   0.404],
                   [ 6.8048,  0.7224,  7.9081, -0.1277],
                   [  0.456,   0.404, -0.1277,   0.997]])
#                          ky     grs       qz      an
Vh_gasp=np.matrix([[ 0.4031, 0.2964, -0.0161, 0.3597],
                   [ 0.2964, 1.8793,   0.023, 0.8438],
                   [ -0.0161,  0.023,  0.0654, 0.0187],
                   [ 0.3597, 0.8438,  0.0187, 0.5571]])
Vh = np.block([[Vh_gabi, np.zeros((4,4))],
               [np.zeros((4,4)), Vh_gasp]])
JDH=np.array([[-1,-1,1,1,0,0,0,0],
              [0,0,0,0,-2,-1,-1,3]])
VdH = np.dot(np.dot(JDH,Vh),JDH.T)
VHlnK = np.block([[VdH,               np.zeros((2, 2))],
                  [np.zeros((2, 2)), VlnK            ]])
Vo = [dataset["Ky"].V, dataset["Gr"].V,
      dataset["Q"].V, dataset["An"].V]
Ho = [dataset["Ky"].H, dataset["Gr"].H,
      dataset["Q"].H, dataset["An"].H]
So = [dataset["Ky"].S, dataset["Gr"].S,
      dataset["Q"].S, dataset["An"].S]
reaction = np.array([-2,-1,-1,3])
ΔVr_gasp = np.dot(reaction, Vo)
ΔHr_gasp = np.dot(reaction, Ho)
ΔSr_gasp = np.dot(reaction, So)
lnKgasp = log(a_an**3/a_grs)
Vo = [dataset["Phl"].V, dataset["Alm"].V,
      dataset["Ann"].V, dataset["Py"].V]
Ho = [dataset["Phl"].H, dataset["Alm"].H,
      dataset["Ann"].H, dataset["Py"].H]
So = [dataset["Phl"].S, dataset["Alm"].S,
      dataset["Ann"].S, dataset["Py"].S]
reaction = np.array([-1,-1,1,1])
ΔVr_gabi = np.dot(reaction, Vo)
ΔHr_gabi = np.dot(reaction, Ho)
ΔSr_gabi = np.dot(reaction, So)
lnKgabi = log((a_ann*a_py)/(a_phl*a_alm))
k1 = (R*lnKgasp-ΔSr_gasp) / \
     (-ΔVr_gasp*(R*lnKgabi-ΔSr_gabi)+  \
      ΔVr_gabi*(R*lnKgasp-ΔSr_gasp))
```

Worked example 7.12 (cont.)

```
k2 = (R*lnKgabi-ΔSr_gabi) / \
     (-ΔVr_gasp*(R*lnKgabi-ΔSr_gabi)+ \
       ΔVr_gabi*(R*lnKgasp-ΔSr_gasp))
k3 = -ΔVr_gasp/(ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) - \
     ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))
k4 = ΔVr_gabi/(ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) - \
     ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))
k5 = R*ΔHr_gasp/(-ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) + \
     ΔVr_gabi*(R*lnKgasp - ΔSr_gasp)) + \
     R*ΔVr_gasp*(ΔHr_gasp*(R*lnKgabi - ΔSr_gabi) - \
     ΔHr_gabi*(R*lnKgasp - ΔSr_gasp)) / \
     (-ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) + \
       ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))**2
k6 = -R*ΔHr_gabi/(-ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) + \
      ΔVr_gabi*(R*lnKgasp - ΔSr_gasp)) - \
      R*ΔVr_gabi*(ΔHr_gasp*(R*lnKgabi - ΔSr_gabi) - \
      ΔHr_gabi*(R*lnKgasp - ΔSr_gasp)) / \
      (-ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) + \
        ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))**2
k7 = -R*ΔVr_gasp*(ΔHr_gasp*ΔVr_gabi - ΔHr_gabi*ΔVr_gasp) / \
     (ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) - \
      ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))**2
k8 = R*ΔVr_gabi*(ΔHr_gasp*ΔVr_gabi - ΔHr_gabi*ΔVr_gasp) / \
     (ΔVr_gasp*(R*lnKgabi - ΔSr_gabi) - \
      ΔVr_gabi*(R*lnKgasp - ΔSr_gasp))**2
J = np.matrix([[k1,k2,k5,k6],
               [k3,k4,k7,k8]])
JV = np.dot(J,VHlnK)
Vpt = np.dot(JV,J.T)
σ_p = (Vpt[0,0])**0.5 # 0.657
σ_t = (Vpt[1,1])**0.5 # 29.158
ρ_pt = Vpt[0,1]/(σ_p*σ_t) # 0.484
```

3. Calculate and plot the uncertainty ellipse (Figure 7.6):

```
fig, ax = plt.subplots()
ax.plot(T, P, color='r', linewidth=1)
ax.plot(Tgasp, Pgasp, color='royalblue', linewidth=1)
ell, rect = error_ellipse(Tint, Pint, Vpt)
ax.add_artist(ell); ax.add_artist(rect)
ax.set_xlabel("Temperature ($^{\circ}$C)")
ax.set_ylabel("Pressure (kbar)")
```

Worked example 7.12 (cont.)

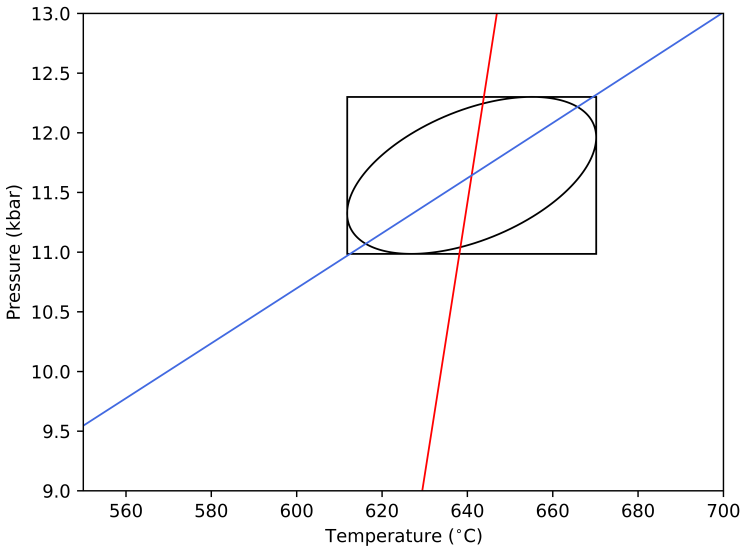

Figure 7.6: *GABI* and *GASP* reactions and an uncertainty ellipse for the $P-T$ estimation of mineral reactions intersection. The ellipse is calculate with the covariance matrix.

Appendix A
Code Listing

```
import pandas as pd
import numpy as np
import sympy as sp
from math import exp, log, isnan
from fractions import Fraction
from scipy import optimize as opt
from scipy.optimize import minimize as min
import scipy.linalg as LA
from scipy.linalg import lu
import matplotlib.pyplot as plt
from matplotlib.patches import Arc
from matplotlib.transforms import Bbox, IdentityTransform,␣
↪→TransformedBbox
```

```
import pandas as pd
import numpy as np
import sympy as sp
from math import exp, log, isnan
from fractions import Fraction
from scipy import optimize as opt
from scipy.optimize import minimize as min
import scipy.linalg as LA
from scipy.linalg import lu
import matplotlib.pyplot as plt
from matplotlib.patches import Arc
from matplotlib.transforms import Bbox, IdentityTransform,␣
↪TransformedBbox
```

```
import pandas as pd
import numpy as np
import sympy as sp
from math import exp, log, isnan
from fractions import Fraction
from scipy import optimize as opt
from scipy.optimize import minimize as min
import scipy.linalg as LA
from scipy.linalg import lu
import matplotlib.pyplot as plt
from matplotlib.patches import Arc
from matplotlib.transforms import Bbox, IdentityTransform,␣
↪TransformedBbox

R = 0.0083144626 #8.31446261815324

# ================================
# Cp data for Grossular (worked example 2.4)
H1_T = np.array([129.79, 135.05, 144.12, 153.68, 162.33, 171.21,
                 180.34, 189.19, 198.00, 206.77, 215.87, 225.46])
H1_Cp = np.array([137.06, 145.46, 159.91, 174.33, 187.38, 200.31,
                  212.63, 219.29, 235.69, 245.32, 255.66, 266.12])
H2_T = np.array([211.48, 217.70, 228.00, 239.30, 250.94, 262.35,
                 272.86, 282.43, 302.55, 312.73, 322.74, 332.62,
                 343.42])
H2_Cp = np.array([249.40, 253.83, 269.30, 280.40, 295.39, 299.14,
                  311.32, 321.73, 335.84, 344.52, 351.46, 358.39,
                  365.14])
H3_T = np.array([11.45, 13.18, 14.92, 16.83, 18.89, 20.92, 22.87,
                 25.03, 27.47, 30.17, 33.34, 36.50, 39.43, 42.54,
                 45.70, 49.39, 53.78])
H3_Cp = np.array([0.074, 0.105, 0.333, 0.457, 0.607, 0.941, 1.223,
                  1.743, 2.316, 3.173, 4.556, 6.248, 7.939, 10.209,
                  12.676, 16.063, 20.166])
H4_T = np.array([56.55, 61.06, 66.45, 71.92, 78.09, 85.26,
                 92.98, 102.09, 111.75, 120.62, 129.78, 139.40])
H4_Cp = np.array([23.905, 29.437, 36.479, 43.57, 52.62, 64.25,
                  76.41, 91.12, 107.02, 122.16, 137.08, 152.26])
K_T = np.array([350.2, 370.1, 390.0, 410.0, 419.9, 429.9, 419.9,
                439.9, 459.8, 479.7, 489.7, 499.7, 509.6, 504.7,
                509.7, 519.7, 539.7, 559.6, 579.6, 589.6, 599.6,
                594.7, 599.7, 609.7, 629.7, 649.6, 659.6, 669.6,
                679.6, 619.8, 629.8, 649.7, 669.7, 689.7, 699.7,
                709.7, 759.7, 769.7, 789.6, 819.6, 829.6, 869.3,
```

(continues on next page)

(continued from previous page)

```
                879.2, 899.0, 908.9, 918.8, 903.7, 918.4, 938.0,
                947.8, 952.7, 962.5, 977.2, 987.1])
K_Cp = np.array([371.1, 379.9, 392.6, 401.0, 403.8, 408.7, 401.8,
                 410.0, 419.6, 424.9, 424.4, 429.7, 430.8, 434.4,
                 438.8, 440.5, 446.9, 450.7, 452.7, 452.9, 451.6,
                 456.7, 458.5, 457.9, 459.9, 462.4, 462.1, 464.0,
                 467.9, 453.6, 454.5, 458.3, 460.4, 462.3, 462.8,
                 465.4, 468.0, 463.4, 465.7, 472.9, 477.6, 494.8,
                 491.4, 495.7, 497.2, 492.1, 480.1, 480.5, 481.2,
                 485.4, 482.2, 481.8, 489.7, 485.6])

# ================================

def load_ds():
    elements = ["Si", "Ti", "Al", "Fe", "Mg", "Mn", "Ca", "Na",
                "K", "O", "H", "C", "Cl", "O2-", "Ni", "Zr", "S",
                "Cu", "Cr"]
    dsContents = []
    with open("../tc-ds62.txt") as dsFile:
        dsContents = dsFile.readlines()
    dsInfo = dsContents[0].split()
    em_count = int(dsInfo[0])
    DS = {}
    for i in range(3, em_count*4, 4):
        dsLine1 = dsContents[i].split()
        dsLine2 = dsContents[i+1].split()
        dsLine3 = dsContents[i+2].split()
        dsLine4 = dsContents[i+3].split()
        em = dsLine1.pop(0)
        # Capitalize the first character
        em = em[:1].upper() + em[1:]
        comp = {}
        totalElements = 0
        for i in range(1, len(dsLine1)-1, 2):
            i_element = int(dsLine1[i])
            content = float(dsLine1[i+1])
            totalElements += content
            comp[elements[i_element-1]] = content
        DS_em = {"comp": comp}
        # H, S, V
        dataArray2 = list(map(float, dsLine2))
        DS_em = DS_em | {'H':dataArray2[0], 'S':dataArray2[1],
                         'V':dataArray2[2]}
```

(continues on next page)

(continued from previous page)

```
        # Cp
        dataArray3 = list(map(float, dsLine3))
        DS_em = DS_em | {'a':dataArray3[0],'b':dataArray3[1],
                         'c':dataArray3[2], 'd':dataArray3[3]}
        # EOS
        dataArray4 = list(map(float, dsLine4))
        DS_em = DS_em | {'alpha': dataArray4[0],
                         'kappa': dataArray4[1],
                         'kappa_p': dataArray4[2],
                         'kappa_pp': dataArray4[3]}
        code = float(dataArray4[4])
        DS_em['flag'] = int(code)
        gases = ['CH4', 'H2', 'CO', 'H2S', 'S2', 'H2O', 'CO2']
        if not(em in gases or code == -1):
            DS_em['theta'] = 10636.0 / (dataArray2[1]* \
                                        1000.0/totalElements+6.44)
        match code:
            case 1.0: # Landau ordering
                tc, s_max, v_max = dataArray4[5:8]
                landau = {'s_max': s_max, 'tc': tc, 'v_max': v_max}
                DS_em['ordering'] = landau
            case 2.0: # Bragg & William ordering
                h,v,wh,wv,n,fac = dataArray4[5:11]
                BW = {'h': h, 'v': v, 'wh': wh,
                      'wv': wv, 'fac': fac, 'n': n}
                DS_em['ordering'] = BW
            case -1.0: # aqueous
                DS_em['cpAq'] = float(dataArray4[5])
            case value if value != 0.0: # melt
                DS_em['flag'] = 5
                DS_em['dK'] = value
        DS[em] = DS_em
    dataset = pd.DataFrame(DS)
    dataset.set_index(dataset.columns[0])
    return dataset

dataset = load_ds()

#====================================
# For calculating Gibbs of H2O
def calc_C_h2o(T):
    cG1 = [0.0,0.0,0.24657688e6,0.51359951e2,0.0,0.0]
    cG2 = [0.0,0.0,0.58638965e0,-0.28646939e-2,0.31375577e-4, 0.0]
```

(continues on next page)

(continued from previous page)

```
    cG3 = [0.0,0.0,-0.62783840e1,0.14791599e-1,
          0.35779579e-3, 0.15432925e-7]
    cG4 = [0.0,0.0,0.0,-0.42719875e0, -0.16325155E-4,0.0]
    cG5 = [0.0,0.0,0.56654978e4,-0.16580167e2,0.76560762e-1,0.0]
    cG6 = [0.0,0.0,0.0,0.10917883e0,0.0,0.0]
    cG7 = [0.38878656e13,-0.13494878e9,0.30916564e6,0.75591105e1,
          0.0,0.0]
    cG8 = [0.0,0.0,-0.65537898e5,0.18810675e3,0.0,0.0]
    cG9 = [-0.14182435e14,0.18165390e9,-0.19769068e6,-0.23530318e2,
          0.0,0.0]
    cG10 = [0.0,0.0,0.92093375e5,0.12246777e3,0.0,0.0]
    cGas = [cG1, cG2, cG3, cG4, cG5, cG6, cG7, cG8, cG9, cG10]
    c1  = cGas[0][0]/T**4 + cGas[0][1]/T**2 + cGas[0][2]/T + \
          cGas[0][3] + cGas[0][4] * T + cGas[0][5] * T**2
    c2  = cGas[1][0]/T**4 + cGas[1][1]/T**2 + cGas[1][2]/T + \
          cGas[1][3] + cGas[1][4] * T + cGas[1][5] * T**2
    c3  = cGas[2][0]/T**4 + cGas[2][1]/T**2 + cGas[2][2]/T + \
          cGas[2][3] + cGas[2][4] * T + cGas[2][5] * T**2
    c4  = cGas[3][0]/T**4 + cGas[3][1]/T**2 + cGas[3][2]/T + \
          cGas[3][3] + cGas[3][4] * T + cGas[3][5] * T**2
    c5  = cGas[4][0]/T**4 + cGas[4][1]/T**2 + cGas[4][2]/T + \
          cGas[4][3] + cGas[4][4] * T + cGas[4][5] * T**2
    c6  = cGas[5][0]/T**4 + cGas[5][1]/T**2 + cGas[5][2]/T + \
          cGas[5][3] + cGas[5][4] * T + cGas[5][5] * T**2
    c7  = cGas[6][0]/T**4 + cGas[6][1]/T**2 + cGas[6][2]/T + \
          cGas[6][3] + cGas[6][4] * T + cGas[6][5] * T**2
    c8  = cGas[7][0]/T**4 + cGas[7][1]/T**2 + cGas[7][2]/T + \
          cGas[7][3] + cGas[7][4] * T + cGas[7][5] * T**2
    c9  = cGas[8][0]/T**4 + cGas[8][1]/T**2 + cGas[8][2] / T + \
          cGas[8][3] + cGas[8][4] * T + cGas[8][5] * T**2
    c10 = cGas[9][0]/T**4 + cGas[9][1]/T**2 + cGas[9][2]/T + \
          cGas[9][3] + cGas[9][4] * T + cGas[9][5] * T**2
    return (c1,c2,c3,c4,c5,c6,c7,c8,c9,c10)

def gasVol(guess, c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,P,T):
    R_cm3 = R*10000 # in cm3 bar /(K mol)
    V = guess[0]
    vol = 1/V + c1 * 1/V**2 - 1/V**2 * \
                              (c3 + 2*c4*1/V+ 3*c5*1/V**2 + \
          4*c6*1/V**3)/(c2 + c3*1/V + c4*1/V**2 + c5*1/V**3 + \
                          c6*1/V**4)**2 + c7*1/V**2*exp(-c8*1/V) + \
          c9*1/V**2*exp(-c10*1/V)
    vol = R_cm3*T*res - P*1000
```

(continues on next page)

(continued from previous page)

```
    return [vol]

def fugGas(V,P,T,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10):
    R_cm3 = R*10000
    A_res = c1/V + 1/(c2+c3/V+c4/V**2+c5/V**3+c6/V**4) - 1/c2 - \
            c7/c8*(exp(-c8/V)-1) - c9/c10 * (exp(-c10/V)-1)
    fug = (A_res + log(1/V) + P*1000*V/(R_cm3*T) + \
           log(R_cm3*T) - 1) * R * T
    return fug

#=========================================

def findQ_NR(P,T,n,H,V,Wh,Wv,fac):
    Q = 0.9999999
    dGdQ = 1
    iterations = 0
    while abs(dGdQ) > 0.0000001 and iterations < 20:
        dGdQ = -H + P * (-V+Wv) - 2*Q*(P*Wv+Wh) + Wh + \
               fac*R*T/(n+1)*(-n*log \
                              ((-Q+1)/(n+1))+n*log((Q+n)/(n+1)) + \
                              n*log((Q*n+1)/(n+1))-n*log \
                              (n*(-Q+1)/(n+1)))
        d2GdQ2 = -2*P*Wv+fac*R*T/(n+1)*(n**2/(Q*n+1)+ \
                                        n/(Q+n)+2*n/(-Q+1))-2*Wh
        Q = Q - dGdQ/d2GdQ2
        iterations += 1
    return Q

def EM_Gibbs_dif(P, T, emData):
    #ds = dataset[em]
    To = 298.15; Po = 0.001; T = T + 273.15
    Ho = emData.H; So = emData.S; Vo = emData.V
    a = emData.a; b = emData.b; c = emData.c; d = emData.d
    Vt = 0
    if isnan(emData.theta): # "H2O"
        (c1,c2,c3,c4,c5,c6,c7,c8,c9,c10) = calc_C_h2o(T)
        x = [20]
        solution = opt.root(gasVol, x,
                            args=(c1,c2,c3,c4,c5,c6,
                                  c7,c8,c9,c10,P,T), method='hybr')
        V = solution.x[0]
        Vt = V/10
        Gv = fugGas(V,P,T,c1,c2,c3,c4,c5,c6,c7,c8,c9,c10)
```

(continues on next page)

(continued from previous page)

```
    else:
        theta = emData.theta
        alpha = emData.alpha; kappa = emData.kappa
        kappa_p = emData.kappa_p; kappa_pp = emData.kappa_pp
        u_o = theta/To
        u = theta/T
        xi_o = (u_o**2)*exp(u_o)/((exp(u_o)-1)**2)
        Pth = alpha*kappa*theta/xi_o*((1/(exp(u)-1))- \
                                      (1/(exp(u_o)-1)))
        a_eos = (1+kappa_p)/(1+kappa_p+kappa*kappa_pp)
        b_eos = kappa_p/kappa - kappa_pp/(1+kappa_p)
        c_eos = (1+kappa_p+kappa*kappa_pp)/(kappa_p**2+\
                                            kappa_p-kappa*kappa_pp)
        Gvp = P*Vo*(1-a_eos*(1+((1+b_eos*(P-Pth))**(1-c_eos))/\
                             (b_eos*P*(c_eos-1))))
        Gvpo = Po*Vo*(1-a_eos*(1+((1+b_eos*(Po-Pth))**(1-c_eos))/\
                               (b_eos*Po*(c_eos-1))))
        Gv = Gvp - Gvpo
        Vt = Vo * (1 - a_eos * (1 - ((1 + b_eos * \
                                      (P - Pth))**(-c_eos))))
    Int_Cp = a*T + b/2*T**2 - c/T + 2*d*T**(0.5) - \
             (a*To + b/2*To**2 - c/To + 2*d*To**(0.5))
    Int_CpT = (a*log(T) + b*T - c/(2*T**2) - 2*d/(T**0.5) \
              - (a*log(To) + b*To - c/(2*To**2) - 2*d/(To**0.5)))
    Gord = 0
    if emData.flag == 1: #"Landau"
        landau = emData.ordering
        Smax = landau['s_max']
        Vmax = landau['v_max']
        Tc_o = landau['tc']
        Tc = Tc_o + (Vmax / Smax) * P
        Qo = ((Tc_o-To)/Tc_o)**(1/4)
        Q4 = ((Tc-T)/Tc_o)
        Q = 0
        if Q4 > 0:
            Q = Q4**(1/4)
        Gord = Smax*Tc_o*(Qo**2 - 1/3*Qo**6) - \
               Smax*(Tc*Q**2 - 1/3*Tc_o*Q**6) - \
               T*Smax*(Qo**2 - Q**2) + P*Vmax*Qo**2
    elif emData.flag == 2: # "BW"
        BW = emData.ordering
        H_BW = BW['h']; V_BW = BW['v']; Wh_BW = BW['wh']
        Wv_BW = BW['wv']; n = BW['n']; fac = BW['fac']
```

(continues on next page)

(continued from previous page)

```
        # Note that this uses the function defined above
        Q = findQ_NR(P,T,n,H_BW,V_BW,Wh_BW,Wv_BW,fac)
        W_BW = Wh_BW + P * Wv_BW
        Tc = 2*W_BW / (R*(1+n))
        Xa_s = (1+n*Q)/(n+1); Xa_sp = (1-Q)/(n+1)
        Xb_s = (n-n*Q)/(n+1); Xb_sp = (n+Q)/(n+1)
        Sord = fac*(-R)* (Xa_s * log(Xa_s) + Xb_s* log(Xb_s) \
               + n*Xa_sp * log(Xa_sp) + n*Xb_sp* log(Xb_sp))
        Hord = H_BW + P*V_BW + Q * ((Wh_BW-H_BW) \
               + P*(Wv_BW-V_BW)) - Q**2 * (Wh_BW+P*Wv_BW)
        Gord = Hord - T * Sord
    dG_dT = -So - Int_CpT
    G = Ho - T*So + Int_Cp - T*Int_CpT + Gv + Gord
    return (Vt, dG_dT, G)

def EM_Gibbs(P, T, emData):
    (_, _, G) = EM_Gibbs_dif(P, T, emData)
    return G

class SSPhaseIdeal:

    def __init__(self, endmembers, sites, vars,
                 prop_eq, ideal_act_eq):
        self.len_em = len(endmembers)
        self.endmembers = endmembers
        self.sites = sites
        self.vars = vars
        self.prop_eq = prop_eq
        self.ideal_act_eq = ideal_act_eq

    def calc_properties(self, P, T, comp):
        self.P = P
        self.T = T
        self.comp = comp
        self.__gibbs()

    def __gibbs(self):
        # Values of variables to be substituted in equations
        subs_vars = []
        for (var_name, var_value) in zip(self.vars, self.comp):
            subs_vars.append((var_name, var_value))
        # proportions of end members
        Xem = np.zeros(self.len_em)
```

(continues on next page)

(continued from previous page)

```
        for i in range(self.len_em):
            em = self.endmembers[i]
            Xem[i] = self.prop_eq[em].subs(subs_vars)
        # Values of site prop. to be substituted in activity eq.
        subs_site_prop = []
        for site in self.sites:
            subs_site_prop.append((site,
                           self.prop_eq[site].subs(subs_vars)))
        G = 0
        Gideal = 0
        act = np.zeros(self.len_em)
        chemicalPotential = np.zeros(self.len_em)
        for i in range(self.len_em):
            em = self.endmembers[i]
            act_eq = self.ideal_act_eq[i]
            act[i] = act_eq.subs(subs_site_prop) # ideal activity
            log_act = log(act[i]) if act[i] > 0 else log(1e-64)
            # Gibbs for end members in dataset
            μo = EM_Gibbs(self.P,self.T, dataset[em.name])
            # Chemical potencial of em in phase
            μ = μo + R*(self.T+273.15)*log_act
            # Gibbs free energy of phase
            G += Xem[i] * μ
            Gideal += Xem[i] * R*(self.T+273.15)*log_act
            chemicalPotential[i] = μ
        self.result = (Xem,act,chemicalPotential,Gideal,G)

class SSPhase:

    def __init__(self, endmembers, sites, vars, prop_eq,
                 ideal_act_eq, w, alphas, dqf, makes = {},
                 rx_ordered = None):
        self.len_em = len(endmembers)
        self.endmembers = endmembers
        self.sites = sites
        self.vars = vars
        self.prop_eq = prop_eq
        self.ideal_act_eq = ideal_act_eq

        self.w = w
        self.alphas = alphas
        self.dqf = dqf
        self.makes = makes
```

(continues on next page)

(continued from previous page)

```
        self.rx_ordered = rx_ordered

    def calc_properties(self, P, T, comp, calc_order = True,
                        μo_MC = None, w_MC = None):
        self.P = P
        self.T = T
        self.comp = comp

        # For Monte Carlo simulations
        self.μo_MC = μo_MC # This is a dictionary
        if w_MC is not None:
            self.w = w_MC

        if self.rx_ordered and calc_order: # Phase with ordered em
            self.order = len(self.rx_ordered)
            orderVars = comp[-self.order:]
            solve_order = opt.root(self.__findOrderState,
                                   orderVars, method='lm', args=())
        else:
            self.__gibbs()

    def __gibbs(self):
        # Values of variables to be substituted in equations
        subs_vars = []
        for (var_name, var_value) in zip(self.vars, self.comp):
            subs_vars.append((var_name, var_value))

        Xem = np.zeros(self.len_em) # prop. of end members
        sum = 0 # sum of alphas times prop. for em
        for i in range(self.len_em):
            em = self.endmembers[i]
            Xem[i] = self.prop_eq[em].subs(subs_vars)
            sum += Xem[i] * self.alphas[i]
        # Values of site prop. to be substituted in activity eq.
        subs_site_prop = []
        for site in self.sites:
            subs_site_prop.append((site,
                           self.prop_eq[site].subs(subs_vars)))
        G = 0
        Gexc = 0
        Gideal = 0
        act = np.zeros(self.len_em)
        chemicalPotential = np.zeros(self.len_em)
```

(continues on next page)

(continued from previous page)

```
        for i in range(self.len_em):
            em = self.endmembers[i]
            act_eq = self.ideal_act_eq[i]
            act[i] = act_eq.subs(subs_site_prop) # ideal activity
            log_act = log(act[i]) if act[i] > 0 else log(1e-64)
            RTLnGamma = 0  # activity coefficient / Gexcess
            idx = 0        # counter for Ws
            for j in range(self.len_em-1):
                kroneckerJ = 1 if j==i else 0
                phiJ = self.alphas[j]*Xem[j]/sum
                for k in range(j+1, self.len_em):
                    kroneckerK = 1 if k==i else 0
                    phiK = self.alphas[k]*Xem[k]/sum
                    asf = 2*self.alphas[i]/ \
                          (self.alphas[j]+self.alphas[k])
                    Wjk = (self.w[idx][0] + \
                           self.w[idx][1]*(self.T+273.15) + \
                           self.w[idx][2]*self.P) * asf
                    idx += 1
                    RTLnGamma += -(kroneckerJ - phiJ) * \
                                 (kroneckerK - phiK) * Wjk
            # Calculate Gibbs for make end members
            if em in self.makes:
                make = self.makes[em]
                μo = 0
                for em_comp in make:
                    if self.μo_MC:
                        μo += self.μo_MC[em_comp.name] * \
                              make[em_comp]
                    else:
                        μo += EM_Gibbs(self.P,self.T,
                                       dataset[em_comp.name]) * \
                                       make[em_comp]
            else:
                if self.μo_MC:
                    μo = self.μo_MC[em.name]
                else:
                    # Gibbs for end members in dataset
                    μo = EM_Gibbs(self.P,self.T, dataset[em.name])

            em_dqf = self.dqf[i]  # Adjust properties using DQF
            if em_dqf:
                μo += em_dqf[0] + em_dqf[1] * (self.T+273.15) + \
```

(continues on next page)

(continued from previous page)

```
                        em_dqf[2] * self.P
            # Chemical potencial of em in phase
            μ = μo + R*(self.T+273.15)*log_act + RTLnGamma
            # Real activity
            act[i] = act[i] * exp(RTLnGamma/(R*(self.T+273.15)))
            G += Xem[i] * μ # Gibbs free energy of phase
            Gexc += Xem[i] * RTLnGamma
            Gideal += Xem[i] * R*(self.T+273.15)*log_act
            chemicalPotential[i] = μ
        self.result = (Xem,act,chemicalPotential,Gideal,Gexc,G)
    # Callback function to solve for order state
    def __findOrderState(self, orderParams):
        residuals = np.zeros(self.order)
        # set composition with orderparams
        self.comp[-self.order:] = orderParams
        self.__gibbs() # calculate Gibbs
        # use reactions to calculate residuals
        for i in range (self.order):
            rx_res = 0
            rx = self.rx_ordered[i]
            for em in rx:
                idx = self.endmembers.index(em)
                coeff = rx[em]
                rx_res += coeff * self.result[2][idx]
            residuals[i] = rx_res
        return residuals

class SSPhase_PT:
    '''
    In this class there are no compositional variables,
    'prop_eq' has equations for calculating
    end member proportions based on the site allocation model.
    'site_eq' has equations for calculating site proportions
    based on phase composition from microprobe data
    '''
    def __init__(self, endmembers, sites, cations, site_eq,
                 prop_eq, ideal_act_eq, w, alphas, dqf,
                 makes = {}, rx_ordered = None):
        self.len_em = len(endmembers)
        self.endmembers = endmembers
        self.cations = cations
        self.sites = sites
        self.site_eq = site_eq
```

(continues on next page)

(continued from previous page)

```
        self.prop_eq = prop_eq
        self.ideal_act_eq = ideal_act_eq

        self.w = w
        self.alphas = alphas
        self.dqf = dqf
        self.makes = makes
        self.rx_ordered = rx_ordered

    def calc_properties(self, P, T, comp, calc_order = True,
                        μo_MC = None, w_MC = None):
        '''
        'comp' is the phase composition from microprobe data
        if phase with ordered end members, this will contain
        also order parameters
        '''
        self.P = P
        self.T = T
        self.comp = comp

        # For Monte Carlo simulations
        self.μo_MC = μo_MC # This is a dictionary
        if w_MC is not None:
            self.w = w_MC
        # Phase has an ordered em
        if self.rx_ordered and calc_order:
            self.order = len(self.rx_ordered)
            orderVars = comp[-self.order:]
            solve_order = opt.root(self.__findOrderState,
                                   orderVars, method='lm', args=())
        else:
            self.__gibbs()

    def __gibbs(self):
        # Values of elem. proportions to be substituted in eq.
        Xp = []
        for (p, value) in zip(self.cations, self.comp):
            Xp.append((p, value))
        # Values of site proportions to be substituted in eq.
        subs_site_prop = []
        for site in self.sites:
            subs_site_prop.append((site,
                                   self.site_eq[site].subs(Xp)))
```

(continues on next page)

(continued from previous page)

```
        Xem = np.zeros(self.len_em)    # prop. of end members
        sum = 0 # sum of alphas times prop. for em

        for i in range(self.len_em):
            em = self.endmembers[i]
            Xem[i] = self.prop_eq[em].subs(subs_site_prop)
            sum += Xem[i] * self.alphas[i]

        G = 0
        Gexc = 0
        Gideal = 0
        act = np.zeros(self.len_em)
        chemicalPotential = np.zeros(self.len_em)
        for i in range(self.len_em):
            em = self.endmembers[i]
            act_eq = self.ideal_act_eq[i]
            act[i] = act_eq.subs(subs_site_prop) # ideal activity
            log_act = log(act[i]) if act[i] > 0 else log(1e-64)
            RTLnGamma = 0 # activity coefficient / Gexcess
            idx = 0 # counter for Ws
            for j in range(self.len_em-1):
                kroneckerJ = 1 if j==i else 0
                phiJ = self.alphas[j]*Xem[j]/sum
                for k in range(j+1, self.len_em):
                    kroneckerK = 1 if k==i else 0
                    phiK = self.alphas[k]*Xem[k]/sum
                    asf = 2*self.alphas[i]/(self.alphas[j]+ \
                                            self.alphas[k])
                    Wjk = (self.w[idx][0] + \
                           self.w[idx][1]*(self.T+273.15) + \
                           self.w[idx][2]*self.P) * asf
                    idx += 1
                    RTLnGamma += -(kroneckerJ - phiJ) * \
                                 (kroneckerK - phiK) * Wjk
            # Calculate Gibbs for make end members
            if em in self.makes:
                make = self.makes[em]
                μo = 0
                for em_comp in make:
                    if self.μo_MC:
                        μo += self.μo_MC[em_comp.name] * \
                              make[em_comp]
```

(continues on next page)

(continued from previous page)

```
                    else:
                        μo += EM_Gibbs(self.P,self.T,
                                       dataset[em_comp.name]) * \
                                       make[em_comp]
            else:
                if self.μo_MC:
                    μo = self.μo_MC[em.name]
                else:
                    # Gibbs for end members in dataset
                    μo = EM_Gibbs(self.P,self.T, dataset[em.name])
            em_dqf = self.dqf[i] # Adjust properties using DQF
            if em_dqf:
                μo += em_dqf[0] + em_dqf[1] * (self.T+273.15) + \
                      em_dqf[2] * self.P
            # Chemical potencial of em in phase
            μ = μo + R*(self.T+273.15)*log_act + RTLnGamma
            # Real activity
            act[i] = act[i] * exp(RTLnGamma/(R*(self.T+273.15)))
            # Gibbs free energy of phase
            G += Xem[i] * μ
            Gexc += Xem[i] * RTLnGamma
            Gideal += Xem[i] * R*(self.T+273.15)*log_act
            chemicalPotential[i] = μ
        self.result = (Xem, act, chemicalPotential,
                       Gideal, Gexc, G)
    # Callback function to solve for order state
    def __findOrderState(self, orderParams):
        residuals = np.zeros(self.order)
        # set composition with orderparams
        self.comp[-self.order:] = orderParams
        self.__gibbs() # calculate Gibbs
        # use reactions to calculate residuals
        for i in range (self.order):
            rx_res = 0
            rx = self.rx_ordered[i]
            for em in rx:
                idx = self.endmembers.index(em)
                coeff = rx[em]
                rx_res += coeff * self.result[2][idx]
            residuals[i] = rx_res
        return residuals

# =======================================
```

(continues on next page)

(continued from previous page)

```
# Activity models
# ========================================

# ================== Plagioclase
#            Mixing sites
#         A           T
#         Ca    Na    Al    Si
# Abh     0     0     1     1
# An      1     1     2     0
#
# ca -> CaA
Abh, An, NaA, CaA, SiT, AlT = sp.symbols('Abh An NaA CaA SiT AlT')
ca = sp.symbols('ca')
prop_eq = {AlT: 0.5*ca + 0.5, CaA: ca, NaA: 1.0 - ca,
           SiT: 0.5 - 0.5*ca, Abh: 1.0 - ca, An: ca}
vars = [ca]
em = [Abh, An]
sites = [CaA, NaA, AlT, SiT]
act_eq = [4 * NaA * AlT * SiT, CaA * AlT**2]
w = [[3.1,0,0]]
alphas = [0.643, 1.0]
dqf = [[], [7.03,-0.00466,0]]
Pl = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)
Pl_i = SSPhaseIdeal(em, sites, vars, prop_eq, act_eq)

# ========================================
# ================== White mica
#                Mixing sites
#        M2A                 T1
#        Mg    Fe    Al      Si    Al
# mu     0     0     1       1     1
# cel    1     0     0       2     0
# fcel   0     1     0       2     0
#
# x -> xFeM2A/(xFeM2A + xMgM2A)
# y -> xAlM2A
Mu, Cel, Fcel = sp.symbols("Mu, Cel, Fcel ")
AlM2A, FeM2A, MgM2A = sp.symbols("AlM2A, FeM2A, MgM2A")
SiT1, AlT1 = sp.symbols("SiT1, AlT1")
x, y = sp.symbols("x, y")
prop_eq = sp.solve([1 - Mu - Cel - Fcel,
                    AlT1 - Mu/2,
                    SiT1 - Mu/2 - Cel - Fcel,
```

(continues on next page)

(continued from previous page)

```
                  AlM2A - Mu,
                  FeM2A - Fcel,
                  MgM2A - Cel,
                  y - AlM2A,
                  x - FeM2A/(FeM2A+MgM2A)],
                  [Mu, Cel, Fcel, AlM2A, FeM2A, MgM2A, SiT1, AlT1])
vars = [x,y]
em = [Mu, Cel, Fcel]
sites = [AlM2A, FeM2A, MgM2A, SiT1, AlT1]
act_eq = [4.0 * AlM2A * SiT1 * AlT1,
          MgM2A * SiT1**2,
          FeM2A * SiT1**2]
w = [[0,0,0.2], [0,0,0.2], [0,0,0]]
alphas = [0.63,0.63,0.63]
dqf = [[], [], []]
WM = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)

# ========================================
# ================== biotite
#                        Mixing sites
#         M3                  M12          T1
#         Mg    Fe    Al      Mg    Fe     Si    Al
# phl     1     0     0       2     0      1     1
# ann     0     1     0       0     2      1     1
# east    0     0     1       2     0      0     2
# obi     0     1     0       2     0      1     1
#
#         Fe
# x -> -----------
#       Fe + Mg
# y = AlM3
#          |    Fe              xFeM12        |
# Q ->3 *  |-----------  -  ------------------ |
#          |  Fe + Mg         xFeM12 + xMgM12  |

Phl, Ann, East, Obi = sp.symbols("Phl, Ann, East, Obi")
MgM3, FeM3, AlM3 = sp.symbols("MgM3, FeM3, AlM3")
MgM12, FeM12, SiT, AlT = sp.symbols("MgM12, FeM12, SiT, AlT")
Mg, Fe, Al, Si, Ti, Mn = sp.symbols("Mg, Fe, Al, Si, Ti, Mn")
x, y, Q = sp.symbols("x, y, Q")
prop_eq = {Phl: -0.666666666666667*Q + x*y - x - y + 1.0,
           Ann: -0.333333333333333*Q + x,
           East: y,
```

(continues on next page)

(continued from previous page)

```
           Obi: Q - x*y,
           FeM3: 0.666666666666667*Q - x*y + x,
           MgM3: -0.666666666666667*Q + x*y - x - y + 1.0,
           AlM3: y,
           MgM12: 0.333333333333333*Q - x + 1.0,
           FeM12: -0.333333333333333*Q + x,
           SiT: 0.5 - 0.5*y,
           AlT: 0.5*y + 0.5}
vars = [x,y,Q]
em = [Phl, Ann, East, Obi]
sites = [FeM3, MgM3, AlM3, MgM12, FeM12, SiT, AlT]
act_eq = [4.0 * MgM3 * MgM12**2.0 * SiT * AlT,
          4.0 * FeM3 * FeM12**2.0 * SiT * AlT,
          AlM3 * MgM12**2.0 * AlT**2.0,
          4.0 * FeM3 * MgM12**2.0 * SiT * AlT]
w = [[12,0,0],[10,0,0],[4,0,0],[15,0,0],[8,0,0],[7,0,0]]
alphas = [1,1,1,1]
dqf = [[], [-3,0,0], [], [-3,0,0]]
rx_ordered = [{Obi: -1, Phl: 2/3, Ann: 1/3}]
makes = {Obi: {Phl: 2/3, Ann: 1/3}}
Bt = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas,
             dqf, makes, rx_ordered)


# ========================================
# ================== biotite model without comp. variables

prop_eq = {Phl: MgM3,
           Ann: FeM12,
           East: AlM3,
           Obi: FeM3 - FeM12}
site_eq = {FeM3: Fe - 2*(Si+Al+Ti+Fe+Mg+Mn-6) * (Fe/(Fe+Mg) - Q/3),
           MgM3: Mg - 2*(Si+Al+Ti+Fe+Mg+Mn-6) * (1 - Fe/(Fe+Mg) + \
                                                 Q/3),
           AlM3: Si + Al - 4,
           FeM12: (Si+Al+Ti+Fe+Mg+Mn-6) * (Fe/(Fe+Mg) - Q/3),
           MgM12: (Si+Al+Ti+Fe+Mg+Mn-6) * (1 - Fe/(Fe+Mg) + Q/3),
           SiT: (Si-2)/2,
           AlT: 1-(Si-2)/2}
cations = [Al, Si, Ti, Fe, Mg, Mn, Q]
Bt_PT = SSPhase_PT(em, sites, cations, site_eq, prop_eq, act_eq,
                   w, alphas, dqf, makes, rx_ordered)
```

(continues on next page)

(continued from previous page)

```
# ========================================
# ================== staurolite
#         Mixing sites
#         X
#         Mg    Fe
# Mst     4     0
# Fst     0     4
#
Mst, Fst, FeX, MgX = sp.symbols('Mst, Fst, FeX, MgX')
x = sp.symbols('x')
prop_eq =  sp.solve([1 - Mst - Fst, MgX - Mst,
                     FeX - Fst, x - FeX/(FeX+MgX)],
                    [Mst, Fst, FeX, MgX])
vars = [x]
em = [Mst, Fst]
sites = [FeX, MgX]
act_eq = [MgX**4, FeX**4]
w = [[16,0,0]]
alphas = [1.0, 1.0]
dqf = [[-8,0,0], []]
St = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)

# ========================================
# ================== garnet FM
#         Mixing sites
#         X
#         Mg    Fe
# Py      3     0
# Alm     0     3
# x -> FeX/(FeX+MgX)
Py, Alm, Gr, Spss = sp.symbols('Py, Alm, Gr, Spss')
FeX, MgX, MnX, CaX = sp.symbols('FeX, MgX, MnX, CaX')
x, c, m = sp.symbols('x, c, m')
prop_eq =  sp.solve([1 - Py - Alm,
                   MgX - Py,
                   FeX - Alm,
                   x - FeX/(FeX+MgX)],
                   [Py, Alm, FeX, MgX])
vars = [x]
em = [Py, Alm]
sites = [FeX, MgX]
act_eq = [MgX**3, FeX**3]
```

(continues on next page)

(continued from previous page)

```
w = [[2.5,0,0]]
alphas = [1.0, 1.0]
dqf = [[], []]
Grt_fm = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)

# ========================================
# ================== garnet MnCFM
#         Mixing sites
#         X
#         Mg    Fe    Ca    Mn
# Py      3     0     0     0
# Alm     0     3     0     0
# Gr      0     0     3     0
# Spss    0     0     0     3
# x -> FeX/(FeX+MgX)
# c -> CaX
# m -> MnX
prop_eq =  sp.solve([1 - Py - Alm - Gr - Spss,
                 MgX - Py,
                 FeX - Alm,
                 MnX - Spss,
                 CaX - Gr,
                 x - FeX/(FeX+MgX),
                 c - CaX,
                 m - MnX],
                 [Py, Alm, Gr, Spss, FeX, MgX, MnX, CaX])
vars = [x, c, m]
em = [Py, Alm, Gr, Spss]
sites = [FeX, MgX, CaX, MnX]
act_eq = [MgX**3, FeX**3, CaX**3, MnX**3]
w = [[2.5,0,0], [31,0,0], [2,0,0], [5,0,0], [2,0,0], [0,0,0]]
alphas = [1.0, 1.0, 2.7, 1.0]
dqf = [[], [], [], []]
Grt = SSPhase(em, sites, vars, prop_eq, act_eq, w, alphas, dqf)

# ========================================
# ================== chlorite
#                         Mixing sites
#         M1           M23      M4       T2
#         Mg  Fe  Al   Mg  Fe   Mg  Fe   Si  Al
# Clin     1   0   0    2   0    2   0    1   1
# Afchl    0   1   0    0   2    1   1    1   1
# Ames     0   0   1    2   0    0   2    1   1
```

(continues on next page)

(continued from previous page)

```
# Daph     0   1   0    2   0    1   1    1   1
# Ochl1    0   1   0    2   0    1   1    1   1
# Ochl2    0   1   0    2   0    1   1    1   1
#
#         Fe
# x -> -----------
#       Fe + Mg
#          |    Fe               xFeM1       |
# Q1 ->3 * |----------  -  ---------------- |
#          |  Fe + Mg         xFeM1 + xMgM1  |
#          |    Fe               xFeM4       |
# Q4 ->3 * |----------  -  ---------------- |
#          |  Fe + Mg         xFeM4 + xMgM4  |
# QAl -> (AlM4 - AlM1)/2
# y   -> (AlM1 + AlM4)/2
Clin, Afchl, Ames, Daph = sp.symbols('Clin, Afchl, Ames, Daph')
Ochl1, Ochl4 = sp.symbols('Ochl1, Ochl4')
FeM1, MgM1, AlM1 = sp.symbols('FeM1, MgM1, AlM1')
FeM23, MgM23 = sp.symbols('FeM23, MgM23')
FeM4, MgM4, AlM4 = sp.symbols('FeM4, MgM4, AlM4')
SiT2, AlT2 = sp.symbols('SiT2, AlT2')
x, Q1, Q4, y, QAl = sp.symbols('x, Q1, Q4, y, QAl')

prop_eq = sp.solve([Clin + Afchl + Ames + Daph + \
                    Ochl1 + Ochl4 - 1,
                    Clin + Afchl + Ochl1 - MgM1,
                    Daph + Ochl4 - FeM1,
                    Ames - AlM1,
                    Clin + Afchl + Ames + Ochl4 - MgM23,
                    Daph + Ochl1 - FeM23,
                    Afchl + Ochl4 - MgM4,
                    Ochl1 - FeM4,
                    Clin + Ames + Daph - AlM4,
                    Clin/2 + Afchl + Daph/2 + Ochl1 + \
                    Ochl4 - SiT2,
                    Clin/2 + Ames + Daph/2 - AlT2,
                    x-(FeM1+4*FeM23+FeM4)/\
                      (FeM1+4*FeM23+FeM4+MgM1+4*MgM23+MgM4),
                    y-(AlM1 + AlM4)/2,
                    QAl - (AlM4 - AlM1)/2,
                    Q1-(x-FeM1/(FeM1+MgM1)),
                    Q4-(x-FeM4/(FeM4+MgM4))],
                   [Clin, Afchl, Ames, Daph, Ochl1, Ochl4,
```

(continues on next page)

(continued from previous page)

```
                    FeM1, MgM1, AlM1, FeM23, MgM23,
                    FeM4, MgM4, AlM4, SiT2, AlT2]);
vars = [x, y, Q1, Q4, QAl]
em = [Clin, Afchl, Ames, Daph, Ochl1, Ochl4]
sites = [FeM1, MgM1, AlM1, FeM23, MgM23,
         FeM4, MgM4, AlM4, SiT2, AlT2]
act_eq = [4 * MgM1 * MgM23**4 * AlM4 * SiT2 * AlT2,
          MgM1 * MgM23**4 * MgM4 * SiT2**2,
          AlM1 * MgM23**4 * AlM4 * AlT2**2,
          4 * FeM1 * FeM23**4 * AlM4 * SiT2 * AlT2,
          MgM1 * FeM23**4 * FeM4 * SiT2**2,
          FeM1 * MgM23**4 * MgM4 * SiT2**2]
w = [[17,0,0], [17,0,0], [20,0,0], [30,0,0], [21,0,0],
     [16,0,0], [37,0,0], [20,0,0], [4,0,0],
     [30,0,0], [29,0,0], [13,0,0],
     [18,0,0], [33,0,0],
     [24,0,0]]
alphas = [1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
dqf = [[], [], [], [], [3,0,0], [2.4,0,0]]
makes = {Ochl1: {Afchl: 1, Clin: -1, Daph: 1},
         Ochl4: {Afchl: 1, Clin: -1/5, Daph: 1/5}}
rx_ordered = [{Ochl1: -1, Afchl: 1, Clin: -1, Daph: 1},
              {Ochl4: -1, Afchl: 1, Clin: -1/5, Daph: 1/5},
              {Clin: 2, Afchl: -1, Ames: -1}]
Chl = SSPhase(em, sites, vars, prop_eq, act_eq, w,
              alphas, dqf, makes, rx_ordered)

# ========================================

class PhaseEquilibria:
    def __init__(self, rock_comp = None):
        self.A = np.array([[3, 1, 0, 3, 0, 0],
                           [3, 1, 3, 0, 0, 0],
                           [1, 1, 0, 0, 0, 0],
                           [7.5, 9, 0, 4, 0, 2],
                           [7.5, 9, 4, 0, 0, 2],
                           [3, 1.5, 0, 0, 0.5, 1],
                           [4, 0.5, 0, 1, 0.5, 1],
                           [4, 0.5, 1, 0, 0.5, 1],
                           [3, 0.5, 0, 3, 0.5, 1],
                           [3, 0.5, 3, 0, 0.5, 1],
                           [2, 1.5, 0, 2, 0.5, 1],
                           [3, 0.5, 1, 2, 0.5, 1],
```

(continues on next page)

(continued from previous page)

```
                            [3, 1, 0, 5, 0, 4],
                            [2, 2, 0, 4, 0, 4],
                            [4, 0, 0, 6, 0, 4],
                            [3, 1, 5, 0, 0, 4],
                            [4, 0, 5, 1, 0, 4],
                            [4, 0, 1, 5, 0, 4],
                            [1, 0, 0, 0, 0, 0],
                            [0, 0, 0, 0, 0, 1]])
        self.phases = [Grt_fm,"Sill",St,WM,Bt,Chl,"Q","H2O"]
        self.phases_em = np.array([2,1,2,3,4,6,1,1])
        self.bounds = [slice(0,2), slice(2,3), slice(3,5),
                       slice(5,8), slice(8,12), slice(12,18),
                       slice(18,19), slice(19,20)]
        self.rock_comp = rock_comp

        # Projection matrix, 4th column will be WM calculated comp
        #                     A   F   M  WM  Qz  H2O
        self.pr = np.array([[0., 0., 0., 0., 1., 0.], # SiO2
                            [1., 0., 0., 0., 0., 0.], # Al2O3
                            [0., 1., 0., 0., 0., 0.], # FeO
                            [0., 0., 1., 0., 0., 0.], # MgO
                            [0., 0., 0., 0., 0., 0.], # K2O
                            [0., 0., 0., 0., 0., 1.]])# H2O

    def calculate(self, calc_type, assemb, param,
                  intensive_vars, fixedP = True,
                  fixed_vars=None, fixed_vars_ix=[],
                  zero_modes_ix = []):
        self.assemb = assemb
        self.total_em = self.phases_em[assemb].sum()
        self.G = np.zeros(len(assemb))
        # construct A_ass
        ix_em = np.hstack([np.ogrid[self.bounds[p]] \
                           for p in assemb])
        A_assemb = self.A[ix_em.astype(int)]
        self.A_assemb = A_assemb
        self.R = self.__calcIndependentReactions(A_assemb)

        #============================Gibbs minimization block
        if calc_type == "Gibbs_min":
            self.P = intensive_vars[0]
            self.T = intensive_vars[1]
            # add Lagrange multipliers
```

(continues on next page)

(continued from previous page)

```
            param_ = np.r_[param, np.full(6,-1000.)]
            opt_result = opt.root(self.__Lagrangian, param_)
            # moles_phases
            moles = opt_result.x[0:(len(self.assemb))]
            vars = opt_result.x[len(self.assemb):-6]
            lagrange_mult = opt_result.x[-6:]
            gibbs = np.dot(self.G, moles)
            self.result = (moles, vars, lagrange_mult, gibbs)

        #============================== PTGrid
        if calc_type == "PTgrid":
            eqIntVars = np.zeros(len(intensive_vars))
            variables = np.zeros((len(intensive_vars),
                                  len(param)-1))
            reactions = []
            if fixedP:
                for i in range(len(intensive_vars)):
                    self.P = intensive_vars[i]
                    opt_result = opt.root(self.__findGridUnivAtP,
                                          param)
                    eqIntVars[i] = opt_result.x[-1]
                    variables[i] = opt_result.x[:-1]
                    reactions.append(\
                        self.__calcIndependentReactions (\
                                                self.phases_comp,
                                                decimals = 3))
            else: # This is for fixed T
                pass
            self.result = (eqIntVars, variables, reactions)
        #============================== PTGrid

        #============================== AFM
        if calc_type == "AFM":
            self.P = intensive_vars[0]; self.T = intensive_vars[1]
            if fixed_vars is not None:
                self.indices = fixed_vars_ix
                self.comp_vars = np.zeros((len(fixed_vars),
                                           len(param)+1))
                self.result = []
                for i in range(len(fixed_vars)):
                    var_i = fixed_vars[i]
                    opt_result = opt.root( \
                                 self.__findTriEquilibritium_AFM,
```

(continues on next page)

(continued from previous page)

```
                                param, args=(var_i))
                    if opt_result.success:
                        self.comp_vars[i] = np.insert( \
                                                opt_result.x,
                                                self.indices[0],
                                                var_i)
                        self.result.append( \
                                            self.__processAFMResult())
            else:
                opt_result = opt.root(\
                          self.__findDivEquilibritium_AFM, param)
                if opt_result.success:
                    self.comp_vars = np.array([opt_result.x])
                    self.result = self.__processAFMResult()
        #============================ AFM

        #============================ Pseudo
        if calc_type == "PTpseudo":
            self.indices = zero_modes_ix
            eqIntVars = np.zeros(len(intensive_vars))
            moles = np.zeros((len(intensive_vars), len(assemb)))
            variables = np.zeros((len(intensive_vars),
                                  len(param)-len(assemb)))
            if fixedP:
                for i in range(len(intensive_vars)):
                    self.P = intensive_vars[i]
                    opt_result = opt.root(\
                           self.__findBoundaryPseudoAtP, param)
                    if opt_result.success:
                        eqIntVars[i] = opt_result.x[-1]
                        moles[i] = np.insert(\
                                    opt_result.x[0:len(assemb)-1],
                                    self.indices[0], 0.0)
                        variables[i] = opt_result.x[\
                                                len(assemb)-1:-1]
            else:
                for i in range(len(intensive_vars)):
                    self.T = intensive_vars[i]
                    opt_result = opt.root(\
                           self.__findBoundaryPseudoAtT, param)
                    if opt_result.success:
                        eqIntVars[i] = opt_result.x[-1]
                        moles[i] = np.insert(\
```

(continues on next page)

(continued from previous page)

```
                                    opt_result.x[0:len(assemb)-1],
                                    self.indices[0], 0.0)
                        variables[i] = opt_result.x[\
                                                    len(assemb)-1:-1]
            self.result = (eqIntVars, moles, variables)
        #============================== Pseudo

    #======== Function for Gibbs Minimization
    def __Lagrangian(self, param):
        moles = param[0:(len(self.assemb))]  # moles_phases
        vars = param[len(self.assemb):-6]
        lambdas = param[-6:]
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_L = self.μ - np.dot(self.A_assemb, lambdas)
        return np.r_[r_mb, r_L]
    #=========== end of Function for Gibbs Minimization

    #============================== Functions for PTgrid
    def __findGridUnivAtP(self, param):
        self.T = param[-1]
        self.__calcPhaseProperties(param)
        return np.dot(self.R, self.μ)

    def __findGridUnivAtT(self, param):
        self.P = param[-1]
        self.__calcPhaseProperties(param)
        return np.dot(self.R, self.μ)

    def __findGridInv(self, param):
        self.__calcPhaseProperties(param)
        pass
    #============================== end of Functions for PTgrid

    #============================== Functions for AFM
    def __findDivEquilibritium_AFM(self, param):
        self.__calcPhaseProperties(param)
        return np.dot(self.R, self.μ)

    def __findTriEquilibritium_AFM(self, param, fixedX):
        param_ = np.insert(param, self.indices[0], fixedX)
        self.__calcPhaseProperties(param_)
        return np.dot(self.R, self.μ)
```

(continues on next page)

(continued from previous page)

```
    def __processAFMResult(self):
        # index of WM in assemblage
        ix_WM = np.where(self.assemb==3)[0][0]
        self.pr[:,3] = self.phases_comp[ix_WM]
        pr_i = LA.inv(self.pr)
        result = []
        for i in range(len(self.assemb)):
            if self.assemb[i] not in np.array([3,6,7]):
                # project
                pr_phase_comp = np.dot(pr_i, self.phases_comp[i])
                ptData = pr_phase_comp/(pr_phase_comp[:3]).sum()
                result.append(ptData.round(decimals=3)) # normalize
        return result
    #============================ end of Functions for AFM

    #============================ Functions for pseudosection
    def __findIntersectionPseudo(self, param):
        # param contains: variables, moles of phases
        # (except  2: intersect) and PT
        self.P = param[-1]
        self.T = param[-2]
        vars = param[0:-(len(self.assemb)+2)]
        moles = np.insert(param[0:(len(self.assemb)-1)],
                          self.indices[0], 0.0)
        moles = np.insert(moles, self.indices[1], 0.0)
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_μ = np.dot(self.R, self.μ)
        return  np.r_[r_mb, r_μ]

    def __findBoundaryPseudoAtT(self, param):
        # param contains: variables, moles of phases
        # (except 1: boundary) and P
        self.P = param[-1]
        moles = np.insert(param[0:(len(self.assemb)-1)],
                          self.indices[0], 0.0)
        vars = param[(len(self.assemb)-1):-1]
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_μ = np.dot(self.R, self.μ)
        return  np.r_[r_mb, r_μ]
```

(continues on next page)

(continued from previous page)

```
    def __findBoundaryPseudoAtP(self, param):
        # param contains: moles of phases
        # (except 1: boundary), variables, and T
        self.T = param[-1]
        moles = np.insert(param[0:(len(self.assemb)-1)],
                          self.indices[0], 0.0)
        vars = param[(len(self.assemb)-1):-1]
        self.__calcPhaseProperties(vars)
        r_mb = self.rock_comp - np.dot(self.phases_comp.T, moles)
        r_μ = np.dot(self.R, self.μ)
        return  np.r_[r_mb, r_μ]
    #============ end of Functions for pseudosection

    #============================== utility Functions
    def __calcPhaseProperties(self, vars):
        self.μ = np.zeros(self.total_em)
        self.X_em = np.zeros(self.total_em)
        self.phases_comp = np.zeros((len(self.assemb), 6))
        ix_em = 0 # index for first end member in phase i
        for i in range(len(self.assemb)):
            p = self.assemb[i]
            em = self.phases_em[p] # number of end members in phase
            A_i = self.A[self.bounds[p]]
            if em == 1:
                self.μ[ix_em] = EM_Gibbs(self.P,self.T,
                                         dataset[self.phases[p]])
                self.G[i] = self.μ[ix_em]
                self.X_em[ix_em] = 1.0
                # composition of pure phase
                self.phases_comp[i,:] = A_i
            else:
                ix_var = ix_em - i
                p_vars = vars[ix_var:ix_var+em-1]
                self.phases[p].calc_properties(\
                                    self.P,self.T,p_vars,False)
                self.μ[ix_em:ix_em+em] = self.phases[p].result[2]
                X_em_p = self.phases[p].result[0]
                self.X_em[ix_em:ix_em+em] = X_em_p
                self.G[i] = self.phases[p].result[5]
                # composition of ss phase
                self.phases_comp[i,:] = np.dot(A_i.T, X_em_p)
            ix_em += em
```

(continues on next page)

(continued from previous page)

```
    def __calcIndependentReactions(self, A, decimals = 15):
        (nr, nc) = A.shape
        A_ = np.pad(A, ((0, 0), (0, nr-nc)),
                    mode='constant', constant_values=0)
        l, u = lu(A_, permute_l = True)
        # components = rank of the matrix
        C = np.linalg.matrix_rank(A_)
        reactions_array = (LA.inv(l)[C:,:]).round(\
                                            decimals=decimals)
        R = np.zeros((len(reactions_array), nr))
        for i in range(len(reactions_array)):
            R[i] = self.__simplifyCoefficients(reactions_array[i])
        return R

    def __simplifyCoefficients(self, x, ld = 1000000):
        max = x.max()
        numerators = np.zeros(len(x), dtype=np.int64)
        denominators = np.zeros(len(x), dtype=np.int64)
        for i in range(len(x)):
            f = Fraction(x[i]/max).limit_denominator(ld)
            numerators[i] = f.numerator
            denominators[i] = f.denominator
        common_denom = np.lcm.reduce(denominators)
        numerators = (numerators * common_denom / \
                      denominators).astype(np.int64)
        common_num = np.gcd.reduce(numerators)
        return numerators/common_num

# ======================================
# ==================== Chapter 7.
# ======================================

# ======================================
def get_oxide_data(components):
    oxides = {"Al2O3" : [101.96128,2,3],
              "CaO" : [56.0794,1,1],
              "FeO" : [71.8464,1,1],
              "H2O" : [18.0152,2,1],
              "K2O" : [94.1954,2,1],
              "MgO" : [40.3044,1,1],
              "MnO" : [70.9374,1,1],
              "Na2O" : [61.97894,2,1],
              "TiO2" : [79.866,1,2],
```

(continues on next page)

(continued from previous page)

```
                "SiO2" : [60.0848,1,2]}
    size = len(components)
    fw = np.zeros(size)
    nc_ox = np.zeros(size)
    no_ox = np.zeros(size)
    for i in range(size):
        ox_data = oxides[components[i]]
        fw[i] = ox_data[0]
        nc_ox[i] = ox_data[1]
        no_ox[i] = ox_data[2]
    return fw, nc_ox, no_ox

# ========================================

# 1% relative uncertainty from KohnSpear91b
def calc_fu(components, w, bO, error_rel = 0.01):
    '''
    Calculates formula unit using oxides weight percent ('w') in a
    'bO' oxygen basis components: array with analyzed oxides; must
    be the same size as 'w' name ('components') and wt% ('w') must
    be in the same order in arrays
    '''
    size = len(components)
    fw, nc_ox, no_ox = get_oxide_data(components)
    S_O = (w / fw * no_ox).sum()
    afu = w / fw * nc_ox * (bO / S_O)
    V_ox = np.diag((w*error_rel)**2)
    J = np.zeros((size,size))
    for i in range(size):
        J[i,:] = no_ox / fw * w[i] * nc_ox[i] * -bO / \
                                                (fw[i] * S_O**2)
        J[i,i] += nc_ox[i] * bO / (fw[i] * S_O)
    V_c = np.dot(np.dot(J,V_ox),J.T)
    return (afu, V_c, J)

# ========================================

def lsq_adj(components, w, bO, Sc, Sc_v = None,
            G = None, error_rel = 0.01):
    '''
    Compute least square adjustment of measured oxide wt% ('w')
    using convertion to formula unit in a 'bO' oxygen basis
    conditions: summation of cations = Sc
```

(continues on next page)

(continued from previous page)

```
    constraint: G = contains the constrain equation, last element
    is the constant in equation.
    Sc_v an array of ones when constraining total cations
    or an array with 1s in the cations to be constrained, last
    element should be 1: this is the cation whose result is
    constrained by condition. For example in Bt, Si+Al=5, leave Si
    as last component and Sc_v should have 1 in the position of Al
    and 1 in the position of Si (last element)
    '''
    afu, Vc, J = calc_fu(components, w, bO, error_rel)
    size = len(w)
    if Sc_v is None: # by default summation of all cations
        Sc_v = np.ones(size)

    A = -1.0 * J
    # last row is the condition equation: 1 parameter is
    # calculated from some combination of other derived observ.
    for i in range(len(w)):
        A[len(w)-1,i] = np.dot(J[:len(w)-1,i], Sc_v[:-1])
    B = np.eye(size)
    f = afu/afu.sum() * Sc
    f[-1] = f[-1] - Sc + np.dot(afu[:-1], Sc_v[:-1])
    f[:-1] = f[:-1] - afu[:-1]

    Q = np.diag((w*error_rel)**2) # assume uncorrelated errors
    Qe = np.dot(A, np.dot(Q,A.T))
    We = np.linalg.inv(Qe)
    N = np.dot(B.T, np.dot(We, B))
    t = np.dot(B.T, np.dot(We, f))
    Ninv = np.linalg.inv(N)
    delta = np.dot(Ninv,t)
    # ====== Adjusted afu - only conditions
    result = afu/afu.sum() * Sc + delta

    if G is not None:
        g_constant = G[-1] # constant
        g = np.dot(G[:-1], result) + g_constant
        C = G[:-1]*-1 # derivatives
        M = np.dot(C, np.dot(Ninv, C.T))
        gCD = g - np.dot(C, delta)
        dD = np.dot(Ninv, np.dot(C.T, 1/M*gCD))
        # ====== Adjusted afu - conditions and constraints
        result = result + dD
```

(continues on next page)

(continued from previous page)

```
    Q_inv = np.linalg.inv(Q)#
    fw, nc_ox, _ = get_oxide_data(components)
    w_adj = result / nc_ox * fw
    w_adj = w_adj / w_adj.sum() * w.sum()
    chi_square = np.dot((w_adj - w).T, np.dot(Q_inv, (w_adj - w)))
    return (result, Vc, chi_square)

# ========================================

def ideal_analysis(components, w, b0, alphas, Sc):
    fw, nc_ox, no_ox = TD_functions.get_oxide_data(components)
    p = w/fw
    Q = np.diag((0.01*p)**2)
    f3 = np.eye(len(w))
    for i in range(len(alphas)):
        theta = Sc[i] * no_ox - b0 * nc_ox * alphas[i]
        Ai = np.dot(theta, f3)
        Fi = np.dot(p,Ai)
        p = p - np.dot(Q, np.dot(Ai.T, Fi))/\
                        np.dot(Ai, np.dot(Q, Ai.T))
    w_adj = p * fw
    Q_inv = np.linalg.inv(Q)
    chi_square = np.dot((p - w/fw).T, np.dot(Q_inv, (p - w/fw)))
    return (w_adj, chi_square)


def error_pl_act(T):
    w = np.array([58.7, 26.1, 6.65, 7.73])
    T = T + 273.15
    result, Vc, _ = lsq_adj(["SiO2", "Al2O3", "CaO", "Na2O"],
                            np.array([58.7, 26.1, 6.65, 7.73]), 8,
                            5, G = np.array([0,1,-1,0,-1]))
    Si, Al, Ca, Na = result
    J = np.array([[0.5, 0, 0, 0], # SiT
              [0, 0.5, 0, 0],     # AlT
              [0, 0, 1, 0],       # Ca
              [0, 0, 0, 1]])      # Na
    Vx = np.dot(J, np.dot(Vc, J.T))
    V = np.zeros((5,5))
    V[:-1,:-1] = Vx
    V[4,4] = 1 # std for W
```

(continues on next page)

(continued from previous page)

```
    AlT = Al/2
    SiT = Si/2 - 1
    W = 3.1

    J11 = 4*AlT*Na*exp(Ca*W*(1 - Na)/(R*T))
    J12 = 4*Na*SiT*exp(Ca*W*(1 - Na)/(R*T))
    J13 = 4*AlT*Na*SiT*W*(1 - Na)*exp(Ca*W*(1 - Na)/(R*T))/(R*T)
    J14 = -4*AlT*Ca*Na*SiT*W*exp(Ca*W*(1 - Na)/(R*T))/(R*T) + \
           4*AlT*SiT*exp(Ca*W*(1 - Na)/(R*T))
    J15 = 4*AlT*Ca*Na*SiT*(1 - Na)*exp(Ca*W*(1 - Na)/(R*T))/(R*T)
    J21 = 0
    J22 = 2*AlT*Ca*exp(Na*W*(1 - Ca)/(R*T))
    J23 = -AlT**2*Ca*Na*W*exp(Na*W*(1 - Ca)/(R*T))/(R*T) + \
           AlT**2*exp(Na*W*(1 - Ca)/(R*T))
    J24 = AlT**2*Ca*W*(1 - Ca)*exp(Na*W*(1 - Ca)/(R*T))/(R*T)
    J25 = AlT**2*Ca*Na*(1 - Ca)*exp(Na*W*(1 - Ca)/(R*T))/(R*T)
    J = np.array([[J11, J12, J13, J14, J15],  # ab
                  [J21, J22, J23, J24, J25]]) # an

    Vact = np.dot(J, np.dot(V, J.T))
    std_ab, atd_an = np.diag(Vact)**0.5
    return(std_ab, atd_an)

def error_grt_act(T):
    w = np.array([37.3, 21.7, 31.7, 4.32, 3.59, 0.98])
    T = T + 273.15
    result, Vc, _ = lsq_adj(["SiO2", "Al2O3", "FeO",
                             "MgO", "MnO", "CaO"],
                            np.array([37.3, 21.7, 31.7, 4.32,
                                      0.98, 3.59]), 12, 8)
    _, _, Fe, Mg, Ca, Mn = result
    Vc = Vc[-4:,-4:]
    Jx = np.array([[1/3, 0, 0, 0],  # MgX
                   [0, 1/3, 0, 0],  # FeX
                   [0, 0, 1/3, 0],  # CaX
                   [0, 0, 0, 1/3]])  # MnX
    Vx = np.dot(Jx, np.dot(Vc, Jx.T))
    V = np.zeros((10,10))
    V[:-6,:-6] = Vx
    V[-6:,-6:] = np.diag(np.ones(6)) # std for W

    MgX = Mg/3
    FeX = Fe/3
```

(continues on next page)

(continued from previous page)

```
    CaX = Ca/3
    MnX = Mn/3

    Wpyalm, Wpygr, Wpyspss, Walmgr = 2.5, 31, 2, 5
    Walmspss, Wgrspss = 2, 0

    J11 = MgX**3*(-CaX*Wpygr - FeX*Wpyalm - \
                  MnX*Wpyspss)*exp((-CaX*FeX*Walmgr - \
                                    CaX*MnX*Wgrspss + \
                                    CaX*Wpygr*(1.0-MgX) - \
                                    FeX*MnX*Walmspss - \
                                    FeX*Wpyalm*(MgX-1.0) + \
                                    MnX*Wpyspss*(1.0-MgX)) / \
                                   (R*T))/(R*T) + \
                   3*MgX**2*exp((-CaX*FeX*Walmgr - \
                                 CaX*MnX*Wgrspss + \
                                 CaX*Wpygr*(1.0-MgX) - \
                                 FeX*MnX*Walmspss - \
                                 FeX*Wpyalm*(MgX-1.0) + \
                                 MnX*Wpyspss*(1.0-MgX))/(R*T))

    J12 = MgX**3*(-CaX*Walmgr - MnX*Walmspss - \
                  Wpyalm*(MgX - 1.0))*exp(\
                                 (-CaX*FeX*Walmgr - \
                                  CaX*MnX*Wgrspss + \
                                  CaX*Wpygr*(1.0-MgX) - \
                                  FeX*MnX*Walmspss - \
                                  FeX*Wpyalm*(MgX-1.0) + \
                                  MnX*Wpyspss*(1.0-MgX))/\
                                                  (R*T))/(R*T)

    J13 = MgX**3*(-FeX*Walmgr - MnX*Wgrspss +
                  Wpygr*(1.0-MgX))*exp((-CaX*FeX*Walmgr - \
                                        CaX*MnX*Wgrspss + \
                                        CaX*Wpygr*(1.0-MgX) - \
                                        FeX*MnX*Walmspss - \
                                        FeX*Wpyalm*(MgX - 1.0) + \
                                        MnX*Wpyspss*(1.0-MgX))/\
                                                    (R*T))/(R*T)

    J14 = MgX**3*(-CaX*Wgrspss - FeX*Walmspss + \
                  Wpyspss*(1.0-MgX))*exp(\
                                    (-CaX*FeX*Walmgr - \
```

(continues on next page)

(continued from previous page)

```
                                    CaX*MnX*Wgrspss + \
                                    CaX*Wpygr*(1.0-MgX) - \
                                    FeX*MnX*Walmspss - \
                                    FeX*Wpyalm*(MgX-1.0) + \
                                    MnX*Wpyspss*(1.0 - MgX))/\
                                                    (R*T))/(R*T)

    J15 = -FeX*MgX**3*(MgX - 1.0)*exp(\
                                   (-CaX*FeX*Walmgr - \
                                    CaX*MnX*Wgrspss + \
                                    CaX*Wpygr*(1.0 - MgX) - \
                                    FeX*MnX*Walmspss - \
                                    FeX*Wpyalm*(MgX - 1.0) + \
                                    MnX*Wpyspss*(1.0 - MgX))/\
                                                    (R*T))/(R*T)

    J16 = CaX*MgX**3*(1.0 - MgX)*exp(\
                                  (-CaX*FeX*Walmgr - \
                                   CaX*MnX*Wgrspss + \
                                   CaX*Wpygr*(1.0 - MgX) - \
                                   FeX*MnX*Walmspss - \
                                   FeX*Wpyalm*(MgX - 1.0) + \
                                   MnX*Wpyspss*(1.0 - MgX))/\
                                                    (R*T))/(R*T)

    J17 = MgX**3*MnX*(1.0 - MgX)*exp(\
                                  (-CaX*FeX*Walmgr - \
                                   CaX*MnX*Wgrspss + \
                                   CaX*Wpygr*(1.0-MgX) - \
                                   FeX*MnX*Walmspss - \
                                   FeX*Wpyalm*(MgX-1.0) + \
                                   MnX*Wpyspss*(1.0-MgX))/\
                                                    (R*T))/(R*T)

    J18 = -CaX*FeX*MgX**3*exp(
                           (-CaX*FeX*Walmgr - CaX*MnX*Wgrspss + \
                            CaX*Wpygr*(1.0-MgX) - \
                            FeX*MnX*Walmspss - \
                            FeX*Wpyalm*(MgX-1.0) + \
                            MnX*Wpyspss*(1.0-MgX))/(R*T))/(R*T)

    J19 = -FeX*MgX**3*MnX*exp(\
                           (-CaX*FeX*Walmgr - CaX*MnX*Wgrspss + \
```

(continues on next page)

(continued from previous page)

```
                        CaX*Wpygr*(1.0-MgX) - \
                        FeX*MnX*Walmspss - \
                        FeX*Wpyalm*(MgX-1.0) + \
                        MnX*Wpyspss*(1.0-MgX))/(R*T))/(R*T)

    J110 = -CaX*MgX**3*MnX*exp(\
                        (-CaX*FeX*Walmgr - CaX*MnX*Wgrspss + \
                         CaX*Wpygr*(1.0-MgX) - \
                         FeX*MnX*Walmspss - \
                         FeX*Wpyalm*(MgX-1.0) + \
                         MnX*Wpyspss*(1.0-MgX))/(R*T))/(R*T)

    J21 = FeX**3*(-CaX*Wpygr - MnX*Wpyspss + \
                  Wpyalm*(1.0-FeX))*exp(
                                  (-CaX*MgX*Wpygr - \
                                   CaX*MnX*Wgrspss + \
                                   CaX*Walmgr*(1.0-FeX) - \
                                   MgX*MnX*Wpyspss + \
                                   MgX*Wpyalm*(1.0 - FeX) + \
                                   MnX*Walmspss*(1.0 - FeX))/\
                                                   (R*T))/(R*T)

    J22 = FeX**3*(-CaX*Walmgr - MgX*Wpyalm - \
                  MnX*Walmspss)*exp(\
                                 (-CaX*MgX*Wpygr - \
                                  CaX*MnX*Wgrspss + \
                                  CaX*Walmgr*(1.0-FeX) - \
                                  MgX*MnX*Wpyspss + \
                                  MgX*Wpyalm*(1.0-FeX) + \
                                  MnX*Walmspss*(1.0-FeX))/\
                                                       (R*T))/(R*T) + \
          3*FeX**2*exp((-CaX*MgX*Wpygr - CaX*MnX*Wgrspss + \
                        CaX*Walmgr*(1.0 - FeX) - \
                        MgX*MnX*Wpyspss + \
                        MgX*Wpyalm*(1.0 - FeX) + \
                        MnX*Walmspss*(1.0 - FeX))/(R*T))

    J23 = FeX**3*(-MgX*Wpygr - MnX*Wgrspss + \
                  Walmgr*(1.0-FeX))*exp(\
                                  (-CaX*MgX*Wpygr - \
                                   CaX*MnX*Wgrspss + \
                                   CaX*Walmgr*(1.0-FeX) - \
                                   MgX*MnX*Wpyspss + \
```

(continues on next page)

(continued from previous page)

```
                                    MgX*Wpyalm*(1.0-FeX) + \
                                    MnX*Walmspss*(1.0-FeX))/\
                                                    (R*T))/(R*T)

    J24 = FeX**3*(-CaX*Wgrspss - MgX*Wpyspss + \
                  Walmspss*(1.0-FeX))*exp(\
                                     (-CaX*MgX*Wpygr - \
                                      CaX*MnX*Wgrspss + \
                                      CaX*Walmgr*(1.0-FeX) - \
                                      MgX*MnX*Wpyspss + \
                                      MgX*Wpyalm*(1.0-FeX) + \
                                      MnX*Walmspss*(1.0-FeX))/\
                                                    (R*T))/(R*T)

    J25 = FeX**3*MgX*(1.0-FeX)*exp(\
                                (-CaX*MgX*Wpygr - \
                                 CaX*MnX*Wgrspss + \
                                 CaX*Walmgr*(1.0-FeX) - \
                                 MgX*MnX*Wpyspss + \
                                 MgX*Wpyalm*(1.0-FeX) + \
                                 MnX*Walmspss*(1.0-FeX))/\
                                                    (R*T))/(R*T)

    J26 = -CaX*FeX**3*MgX*exp((-CaX*MgX*Wpygr - \
                                CaX*MnX*Wgrspss + \
                                CaX*Walmgr*(1.0-FeX) - \
                                MgX*MnX*Wpyspss + \
                                MgX*Wpyalm*(1.0-FeX) + \
                                MnX*Walmspss*(1.0-FeX))/(R*T))/(R*T)

    J27 = -FeX**3*MgX*MnX*exp((-CaX*MgX*Wpygr - CaX*MnX*Wgrspss + \
                                CaX*Walmgr*(1.0-FeX) - \
                                MgX*MnX*Wpyspss + \
                                MgX*Wpyalm*(1.0-FeX) + \
                                MnX*Walmspss*(1.0-FeX))/(R*T))/(R*T)

    J28 = CaX*FeX**3*(1.0 - FeX)*exp((-CaX*MgX*Wpygr - \
                                       CaX*MnX*Wgrspss + \
                                       CaX*Walmgr*(1.0 - FeX) - \
                                       MgX*MnX*Wpyspss + \
                                       MgX*Wpyalm*(1.0 - FeX) + \
                                       MnX*Walmspss*(1.0 - FeX))/\
                                                       (R*T))/(R*T)
```

(continues on next page)

(continued from previous page)

```
J29 = FeX**3*MnX*(1.0-FeX)*exp((-CaX*MgX*Wpygr - \
                                CaX*MnX*Wgrspss + \
                                CaX*Walmgr*(1.0-FeX) - \
                                MgX*MnX*Wpyspss + \
                                MgX*Wpyalm*(1.0-FeX) + \
                                MnX*Walmspss*(1.0 - FeX))/\
                                              (R*T))/(R*T)

J210 = -CaX*FeX**3*MnX*exp((-CaX*MgX*Wpygr - \
                            CaX*MnX*Wgrspss + \
                            CaX*Walmgr*(1.0-FeX) - \
                            MgX*MnX*Wpyspss + \
                            MgX*Wpyalm*(1.0-FeX) + \
                            MnX*Walmspss*(1.0-FeX))/\
                                          (R*T))/(R*T)

J31 = CaX**3*(-FeX*Wpyalm - MnX*Wpyspss + \
              Wpygr*(1.0-CaX))*exp(\
                           (-FeX*MgX*Wpyalm - \
                            FeX*MnX*Walmspss + \
                            FeX*Walmgr*(1.0-CaX) - \
                            MgX*MnX*Wpyspss + \
                            MgX*Wpygr*(1.0-CaX) + \
                            MnX*Wgrspss*(1.0-CaX))/\
                                            (R*T))/(R*T)

J32 = CaX**3*(-MgX*Wpyalm - MnX*Walmspss + \
              Walmgr*(1.0-CaX))*exp(\
                            (-FeX*MgX*Wpyalm - \
                             FeX*MnX*Walmspss + \
                             FeX*Walmgr*(1.0-CaX) - \
                             MgX*MnX*Wpyspss + \
                             MgX*Wpygr*(1.0-CaX) + \
                             MnX*Wgrspss*(1.0 - CaX))/\
                                             (R*T))/(R*T)

J33 = CaX**3*(-FeX*Walmgr - MgX*Wpygr - \
              MnX*Wgrspss)*exp(
                          (-FeX*MgX*Wpyalm - \
                           FeX*MnX*Walmspss + \
                           FeX*Walmgr*(1.0-CaX) - \
                           MgX*MnX*Wpyspss + \
```

(continues on next page)

(continued from previous page)

```
                                MgX*Wpygr*(1.0-CaX) + \
                                MnX*Wgrspss*(1.0-CaX))/\
                                                    (R*T))/(R*T) + \
      3*CaX**2*exp((-FeX*MgX*Wpyalm - \
                    FeX*MnX*Walmspss + \
                    FeX*Walmgr*(1.0-CaX) - \
                    MgX*MnX*Wpyspss + \
                    MgX*Wpygr*(1.0-CaX) + \
                    MnX*Wgrspss*(1.0-CaX))/(R*T))

J34 = CaX**3*(-FeX*Walmspss - MgX*Wpyspss + \
              Wgrspss*(1.0-CaX))*exp(
                                  (-FeX*MgX*Wpyalm - \
                                   FeX*MnX*Walmspss + \
                                   FeX*Walmgr*(1.0-CaX) - \
                                   MgX*MnX*Wpyspss + \
                                   MgX*Wpygr*(1.0-CaX) + \
                                   MnX*Wgrspss*(1.0-CaX))/\
                                                      (R*T))/(R*T)

J35 = -CaX**3*FeX*MgX*exp((-FeX*MgX*Wpyalm - \
                           FeX*MnX*Walmspss + \
                           FeX*Walmgr*(1.0-CaX) - \
                           MgX*MnX*Wpyspss + \
                           MgX*Wpygr*(1.0-CaX) + \
                           MnX*Wgrspss*(1.0-CaX))/(R*T))/(R*T)

J36 = CaX**3*MgX*(1.0-CaX)*exp((-FeX*MgX*Wpyalm - \
                                FeX*MnX*Walmspss + \
                                FeX*Walmgr*(1.0-CaX) - \
                                MgX*MnX*Wpyspss + \
                                MgX*Wpygr*(1.0-CaX) + \
                                MnX*Wgrspss*(1.0-CaX))/\
                                                    (R*T))/(R*T)

J37 = -CaX**3*MgX*MnX*exp((-FeX*MgX*Wpyalm - \
                           FeX*MnX*Walmspss + \
                           FeX*Walmgr*(1.0-CaX) - \
                           MgX*MnX*Wpyspss + \
                           MgX*Wpygr*(1.0-CaX) + \
                           MnX*Wgrspss*(1.0-CaX))/\
                                                    (R*T))/(R*T)
```

(continues on next page)

(continued from previous page)

```
    J38 = CaX**3*FeX*(1.0-CaX)*exp((-FeX*MgX*Wpyalm - \
                                    FeX*MnX*Walmspss + \
                                    FeX*Walmgr*(1.0-CaX) - \
                                    MgX*MnX*Wpyspss + \
                                    MgX*Wpygr*(1.0-CaX) + \
                                    MnX*Wgrspss*(1.0-CaX))/\
                                                    (R*T))/(R*T)

    J39 = -CaX**3*FeX*MnX*exp((-FeX*MgX*Wpyalm - \
                               FeX*MnX*Walmspss + \
                               FeX*Walmgr*(1.0-CaX) - \
                               MgX*MnX*Wpyspss +\
                               MgX*Wpygr*(1.0-CaX) + \
                               MnX*Wgrspss*(1.0-CaX))/(R*T))/(R*T)

    J310 = CaX**3*MnX*(1.0-CaX)*exp((-FeX*MgX*Wpyalm - \
                                     FeX*MnX*Walmspss + \
                                     FeX*Walmgr*(1.0-CaX) - \
                                     MgX*MnX*Wpyspss + \
                                     MgX*Wpygr*(1.0- aX) + \
                                     MnX*Wgrspss*(1.0-CaX))/\
                                                     (R*T))/(R*T)

    J41 = MnX**3*(-CaX*Wpygr - FeX*Wpyalm + \
                  Wpyspss*(1.0-MnX))*exp((-CaX*FeX*Walmgr - \
                                          CaX*MgX*Wpygr + \
                                          CaX*Wgrspss*(1.0-MnX) - \
                                          FeX*MgX*Wpyalm + \
                                          FeX*Walmspss*(1.0-MnX) + \
                                          MgX*Wpyspss*(1.0 - MnX))/\
                                                          (R*T))/(R*T)

    J42 = MnX**3*(-CaX*Walmgr - MgX*Wpyalm + \
                  Walmspss*(1.0-MnX))*exp(\
                                         (-CaX*FeX*Walmgr \
                                          - CaX*MgX*Wpygr + \
                                          CaX*Wgrspss*(1.0-MnX) - \
                                          FeX*MgX*Wpyalm + \
                                          FeX*Walmspss*(1.0-MnX) + \
                                          MgX*Wpyspss*(1.0-MnX))/\
                                                          (R*T))/(R*T)

    J43 = MnX**3*(-FeX*Walmgr - MgX*Wpygr + \
```

(continues on next page)

(continued from previous page)

```
                Wgrspss*(1.0-MnX))*exp((-CaX*FeX*Walmgr - \
                                        CaX*MgX*Wpygr + \
                                        CaX*Wgrspss*(1.0-MnX) - \
                                        FeX*MgX*Wpyalm + \
                                        FeX*Walmspss*(1.0-MnX) + \
                                        MgX*Wpyspss*(1.0-MnX))/\
                                                    (R*T))/(R*T)

    J44 = MnX**3*(-CaX*Wgrspss - FeX*Walmspss - \
                MgX*Wpyspss)*exp((-CaX*FeX*Walmgr - \
                                    CaX*MgX*Wpygr + \
                                    CaX*Wgrspss*(1.0-MnX) - \
                                    FeX*MgX*Wpyalm + \
                                    FeX*Walmspss*(1.0-MnX) + \
                                    MgX*Wpyspss*(1.0-MnX))/\
                                                (R*T))/(R*T) + \
         3*MnX**2*exp((-CaX*FeX*Walmgr - CaX*MgX*Wpygr + \
                       CaX*Wgrspss*(1.0-MnX) - FeX*MgX*Wpyalm + \
                       FeX*Walmspss*(1.0-MnX) +\
                       MgX*Wpyspss*(1.0-MnX))/(R*T))

    J45 = -FeX*MgX*MnX**3*exp((-CaX*FeX*Walmgr - CaX*MgX*Wpygr + \
                               CaX*Wgrspss*(1.0-MnX) - \
                               FeX*MgX*Wpyalm + \
                               FeX*Walmspss*(1.0-MnX) + \
                               MgX*Wpyspss*(1.0-MnX))/(R*T))/(R*T)

    J46 = -CaX*MgX*MnX**3*exp((-CaX*FeX*Walmgr - CaX*MgX*Wpygr + \
                               CaX*Wgrspss*(1.0-MnX) - \
                               FeX*MgX*Wpyalm + \
                               FeX*Walmspss*(1.0-MnX) + \
                               MgX*Wpyspss*(1.0-MnX))/(R*T))/(R*T)

    J47 = MgX*MnX**3*(1.0-MnX)*exp((-CaX*FeX*Walmgr - \
                                    CaX*MgX*Wpygr + \
                                    CaX*Wgrspss*(1.0-MnX) - \
                                    FeX*MgX*Wpyalm + \
                                    FeX*Walmspss*(1.0-MnX) + \
                                    MgX*Wpyspss*(1.0-MnX))/\
                                                (R*T))/(R*T)

    J48 = -CaX*FeX*MnX**3*exp((-CaX*FeX*Walmgr - CaX*MgX*Wpygr + \
                               CaX*Wgrspss*(1.0-MnX) - \
```

(continues on next page)

(continued from previous page)

```
                                FeX*MgX*Wpyalm + \
                                FeX*Walmspss*(1.0-MnX) + \
                                MgX*Wpyspss*(1.0-MnX))/(R*T))/(R*T)

    J49 = FeX*MnX**3*(1.0-MnX)*exp((-CaX*FeX*Walmgr - \
                                    CaX*MgX*Wpygr + \
                                    CaX*Wgrspss*(1.0-MnX) - \
                                    FeX*MgX*Wpyalm + \
                                    FeX*Walmspss*(1.0-MnX) + \
                                    MgX*Wpyspss*(1.0-MnX))/\
                                                    (R*T))/(R*T)

    J410 = CaX*MnX**3*(1.0-MnX)*exp((-CaX*FeX*Walmgr - \
                                     CaX*MgX*Wpygr + \
                                     CaX*Wgrspss*(1.0-MnX) - \
                                     FeX*MgX*Wpyalm + \
                                     FeX*Walmspss*(1.0-MnX) + \
                                     MgX*Wpyspss*(1.0-MnX))/\
                                                    (R*T))/(R*T)

    # First 4 columns for MgX  FeX  CaX  MnX
    # last colmumns for Ws pyalm pygr pyspss almgr almspss grspss
    # row order is {Py}, {Alm}, {Gr}, {Spss}
    J = np.array([[J11,J12,J13,J14,J15,J16,J17,J18,J19,J110],
                  [J21,J22,J23,J24,J25,J26,J27,J28,J29,J210],
                  [J31,J32,J33,J34,J35,J36,J37,J38,J39,J310],
                  [J41,J42,J43,J44,J45,J46,J47,J48,J49,J410]])
    Vact = np.dot(J, np.dot(V, J.T))
    std_py, std_alm, std_gr, std_spss = np.diag(Vact)**0.5
    return (std_py, std_alm, std_gr, std_spss)
```

References

Albarède, F. (1996). *Introduction to geochemical modeling*. Cambridge University Press.

Anderson, G. M. (1995). Is there alkali-aluminum complexing at high temperatures and pressures? *Geochimica et Cosmochimica Acta, 59*(11). 2155–2161. https://doi.org/10.1016/0016-7037(95)00097-J

Anderson, G. M., Castet, S., Schott, J., & Mesmer, R. E. (1991). The density model for estimation of thermodynamic parameters of reactions at high temperatures and pressures. *Geochimica et Cosmochimica Acta, 55*(7), 1769–1779. https://doi.org/10.1016/0016-7037(91)90022-W

Anderson, J. L. (1996). Status of thermobarometry in granitic batholiths. *Transactions of the Royal Society of Edinburgh: Earth Sciences, 87* (1-2), 125–138. https://doi.org/10.1017/S0263593300006544

Anderson, J. L., Barth, A. P., Wooden, J. L., & Mazdab, F. (2008). Thermometers and thermobarometers in granitic systems. In K. D. Putirka & F. J. Tepley (Eds.), *Minerals, Inclusions and Volcanic Processes* (RiMG Vol. 69, pp. 121–142). Mineralogical Society of America. https://doi.org/10.2138/rmg.2008.69.4

Angeli, C., Cimiraglia, R., Dallo, F., Guareschi, R., & Tenti, L. (2013). Dependence of the population on the temperature in the boltzmann distribution: A simple relation involving the average energy. *Journal of Chemical Education, 90*(12), 1639–1644. http://dx.doi.org/10.1021/ed300886j

Ashworth, J. R., Sheplev, V. S., Khlestov, V. V., & Ananyev, V. A. (2004). An analysis of uncertainty in non-equilibrium and equilibrium geothermobarometry. *Journal of Metamorphic Geology, 22*(9), 811–824. https://doi.org/10.1111/j.1525-1314.2004.00552.x

Asimow, P. D., & Ghiorso, M. S. (1998). Algorithmic modifications extending melts to calculate subsolidus phase relations. *American Mineralogist, 83*(9-10), 1127–1132. https://doi.org/10.2138/am-1998-9-1022

Berman, R. G. (1988). Internally-consistent thermodynamic data for minerals in the system Na_2O-K_2O-CaO-MgO-FeO-Fe_2O_3-Al_2O_3-SiO_2-TiO_2-H_2O-CO_2. *Journal of Petrology, 29*(2), 445–522. https://doi.org/10.1093/petrology/29.2.445

Berman, R. G. (1991). Thermobarometry using multi-equilibrium calculations: A new technique, with petrological applications. *Canadian Mineralogist, 29*(4), 833–855.

Berman, R. G. (2007). *winTWQ (version 2.3): A software package for performing internally-consistent thermobarometric calculations*. (Open File 5462). Geological Survey of Canada.

Berman, R. G., & Brown, T. H. (1985). Heat capacity of minerals in the system Na_2O-K_2O-CaO-MgO-FeO-Fe_2O_3-Al_2O_3-SiO_2-TiO_2-H_2O-CO_2: representation, estimation, and high temperature extrapolation. *Contributions to Mineralogy and Petrology, 89*, 168–183. https://doi.org/10.1007/BF00379451

Berman, R. G., Engi, M., Greenwood, H. J., & Brown, T. H. (1986). Derivation of internally-consistent thermodynamic data by the technique of mathematical programming: a review with application to the system MgO-SiO_2-H_2O. *Journal of Petrology, 27*(6), 1331–1364. https://doi.org/10.1093/petrology/27.6.1331

Beyer, C., Frost, D. J., & Miyajima, N. (2015). Experimental calibration of a garnet-clinopyroxene geobarometer for mantle eclogites. *Contributions to Mineralogy and Petrology, 169*(18), 1–21. https://doi.org/10.1007/s00410-015-1113-z

Bohlen, S. R., & Lindsley, D. H. (1987). Thermometry and barometry of igneous and metamorphic rocks. *Annual Reviews of Earth and Planetary Sciences, 15*, 397–420. https://doi.org/10.1146/annurev.ea.15.050187.002145

Bohlen, S. R., Wall, V. J., & Boettcher, A. L. (1983). Geobarometry in Granulites. In S. K. Saxena (Ed.), *Kinetics and Equilibrium in Mineral Reactions* (Advances in Physical Geochemistry Vol. 3, pp. 141–171). Springer-Verlag. https://doi.org/10.1007/978-1-4612-5587-1_5

Bowen, N. L. (1940). Progressive metamorphism of siliceous limestone and dolomites. *Journal of Geology, 48*(3), 225–274. https://doi.org/10.1086/624885

Bragg, W. L., & Williams, E. J. (1934). The effect of thermal agitation on atomic arrangement in alloys. *Proceedings of the Royal Society of London, Series A, 145*, 699–730. https://doi.org/10.1098/rspa.1934.0132

Bragg, W. L., & Williams, E. J. (1935). The effect of thermal agitation on atomic arrangement in alloys ii. *Proceedings of the Royal Society of London, Series A, 151*, 540–566. https://doi.org/10.1098/rspa.1935.0165

Carpenter, M. A. (1992). Thermodynamics of phase transitions in minerals: A macroscopic approach. In G. D. Price & N. L. Ross (Eds.), *The stability of minerals* (The Mineralogical Society Series Vol. 3, pp. 172–215). Springer. https://doi.org/10.1007/978-0-585-27578-9_5

Carpenter, M. A., Powell, R., & Salje, E. K. H. (1994). Thermodynamics of nonconvergent cation ordering in minerals: I. An alternative approach. *American Mineralogist, 79*(11-12), 1053–1067.

Carpenter, M. A., & Putnis, A. (1986). Cation order and disorder during crystal growth: some implications for natural mineral assemblages. In A. B. Thompson & D. C. Rubie (Eds.), *Metamorphic reactions, Kinetics, textures and deformation* (Advances in Physical Geochemistry Vol. 4, pp 1–26). Springer. https://doi.org/10.1007/978-1-4612-5066-1_1

Carson, C. J., & Powell, R. (1997). Garnet-orthopyroxene geothermometry and geobarometry: Error propagation and equilibration effects. *Journal of Metamorphic Geology, 15*(6), 679–686. https://doi.org/10.1111/j.1525-1314.1997.00686.x

Carter, J. N. (2004). Using bayesian statistics to capture the effects of modelling errors in inverse problems. *Mathematical Geology, 36*, 187–216. https://doi.org/10.1023/B:MATG.0000020470.51595.6d

Cemič, L., & Dachs, E. (2006). Heat capacity of ferrosilite, $Fe_2Si_2O_6$. *Physics and Chemistry of Minerals, 33*, 457–464. https://doi.org/10.1007/s00269-006-0090-1

Chatterjee, N. D. (1991). *Applied mineralogical thermodynamics. Selected topics*. Springer-Verlag.

Chatterjee, N. J., Krüger, R., Haller, G., & Olbricht, W. (1998). The bayesian approach to an internally consistent thermodynamic database: Theory, database, and generation of phase diagrams. *Contributions to Mineralogy and Petrology, 133*(1), 149–168. https://doi.org/10.1007/s004100050444

Connolly, J. A. D. (2005). Computation of phase equilibria by linear programming: a tool for geodynamic modeling and its application to subduction zone decarbonation. *Earth and Planetary Science Letters, 236*(1–2), 524–541. https://doi.org/10.1016/j.epsl.2005.04.033

Darken, L. S. (1950). Application of the Gibbs-Duhem equation to ternary and multicomponent systems. *Journal of the American Chemical Society, 72*(7), 2909–2914. https://doi.org/10.1021/ja01163a030

Darken, L. S. (1967). Thermodynamics of binary metallic solutions. *Transactions of the metallurgical society of AIME, 239*, 80–89.

de Capitani, C., & Brown, T. H. (1987). The computation of chemical equilibrium in complex systems containing non-ideal solutions. *Geochimica et Cosmochimica Acta, 51*(10), 2639–2652. https://doi.org/10.1016/0016-7037(87)90145-1

de Capitani, C., & Petrakakis, K. (2010). The computation of equilibrium assemblage diagrams with theriak/domino software. *American Mineralogist, 95*(7), 1006–1016. https://doi.org/10.2138/am.2010.3354

Dolejš, D. (2013). Thermodynamics of aqueous species at high temperatures and pressures: Equations of state and transport theory In A. Stefánsson, T. Driesner & P. Bénézeth (Eds.), *Thermodynamics of Geothermal Fluids* (RiMG Vol. 76, pp.35–79). Mineralogical Society of America. https://doi.org/10.2138/rmg.2013.76.3

Dollase, W. A., & Newman, W. I. (1984). Statistically most probable stoichiometric formulae. *American Mineralogist, 69*, 553–556.

Engel, T., & Reid, P. (2013). *Thermodynamics, Statistical Thermodynamics, & Kinetics*. (3rd ed.) Pearson.

Engi, M. (1992). Thermodynamic data for minerals: A critical assessment. In G. D. Price & N. L. Ross (Eds.), *The stability of minerals* (Mineralogical Society Series Vol. 3, pp. 267–328). Kluwer Academic Publishers. https://doi.org/10.1007/978-0-585-27578-9_8

Essene, E. J. (1982). Geologic thermometry and barometry. In J. M. Ferry (Ed.), *Characterization of Metamorphism through Mineral Equilibria* (RiMG Vol. 10, pp. 153–206). Mineralogical Society of America. https://doi.org/10.1515/9781501508172-009

Essene, E. J. (1989). The current status of thermobarometry in metamorphic rocks. In J. S. Daly, R. A. Cliff, & B. W. D. Yardley (Eds.), *Evolution of Metamorphic Belts* (GSL Special Publications Vol. 43, pages 1–44). Geological Society of London. https://doi.org/10.1144/GSL.SP.1989.043.01.02

Evans, T. P. (2004). A method for calculating effective bulk composition modification due to crystal fractionation in garnet-bearing schist:

Implications for isopleth thermobarometry. *Journal of Metamorphic Geology, 22*(6), 547–557. https://doi.org/10.1111/j.1525-1314.2004.00532.x

Fischler, M. A., & Bolles, R. C. (1981). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM, 24*(6), 381–395. https://doi.org/10.1145/358669.358692

Fonarev, V. I., Graphchikov, A. A., & Konilov, A. N. (1991). A consistent system of geothermometers for metamorphic complexes. *International Geology Review, 33*(8), 743–783. https://doi.org/10.1080/00206819109465723

Freund, J., & Ingalls, R. (1989). Inverted isothermal equations of state and determination of B_0, B_0' and B_0". *Journal of Physics and Chemistry of Solids, 50*(3), 263–268. https://doi.org/10.1016/0022-3697(89)90486-1

Gautam, R., & Seider, W. D. (1979). Computation of phase and chemical equilibrium: Part I. Local and constrained minima in Gibbs free energy. *AIChE Journal, 25*(6), 991–999. https://doi.org/10.1002/aic.690250610

Ghiorso, M. S. (1985). Chemical mass transfer in magmatic processes: I. Thermodynamic relations and numerical algorithms. *Contributions to Mineralogy and Petrology, 90*(2-3), 107–120. https://doi.org/10.1007/BF00378254

Ghiorso, M. S. (1994). Algorithms for the estimation of phase stability in heterogeneous thermodynamic systems. *Geochimica et Cosmochimica Acta, 58*(24), 5489–5501. https://doi.org/10.1016/0016-7037(94)90245-3

Ghiorso, M. S., Hirschmann, M. M., Reiners, P. W., & Kress, V. C. (2002). The pMELTS: A revision of MELTS for improved calculation of phase relations and major element partitioning related to partial melting of the mantle to 3 Gpa. *Geochemistry, Geophysics, Geosystems, 3*(5):1–35. https://doi.org/10.1029/2001GC000217

Giaramita, M. J., & Day, H. W. (1990). Error propagation in calculations of structural formulas. *American Mineralogist, 75*, 170–182.

Goldsmith, J. R. (1980). The melting and breakdown reactions of anorthite at high pressures and temperatures. *American Mineralogist, 65*, 272–284.

Gordon, S., & McBride, B. J. (1994). *Computer program for calculation of complex chemical equilibrium compositions and applications. Part 1: Analysis* (NASA Reference Publication 1311). National Aeronautics and Space Administration.

Gordon, T. M. (1998). WEBINVEQ thermobarometry: An experiment in providing interactive scientific, software on the world wide web. *Computers Geosciences, 24*(1), 43–49. https://doi.org/10.1016/S0098-3004(97)00079-4

Guidotti, C. V. (1984). Micas in metamorphic rocks. In S. W. Bailey (Ed.), *Micas* (RiMG Vol. 13, pp. 357–468). Mineralogical Society of America. https://doi.org/10.1515/9781501508820-014

Harley, S. L. (1984). An experimental study of the partitioning of Fe and Mg between garnet and orthopyroxene. *Contributions to Mineralogy and Petrology, 86*, 359–373. https://doi.org/10.1007/BF01187140

Haselton, H. T., & Westrum, E. F. (1980). Low-temperature heat capacities of synthetic pyrope, grossular, and pyrope$_{60}$grossular$_{40}$. *Geochimica et Cosmochimica Acta, 44*, 701–709. https://doi.org/10.1016/0016-7037(80)90159-3

Helgeson, H. C., Delany, J. M., Nesbitt, H. W., & Bird, D. K. (1978). Summary and critique of the thermodynamic properties of rock forming minerals. *American Journal of Science, 278-A*, 1–229.

Hensen, B. H. (1971). Theoretical phase relations involving cordierite and garnet in the system MgO-FeO-Al_2O_3-SiO_2. *Contributions to Mineralogy and Petrology, 33*, 191–214. https://doi.org/10.1007/BF00374063

Hodges, J. S., & Dewar, J. A. (1992). *Is it you or your model talking? A framework for model validation.* (Report No. R-4114-A/AF/OSD). RAND Corporation.

Holdaway, M. J., & Mukhopadhyay, B. (1993). Geothermobarometry in pelitic schists: A rapidly evolving field. *American Mineralogist, 78*, 681–693.

Holland, T. (1981). Thermodynamic analysis of simple mineral systems. In R. C. Newton, A. Navrotsky, & B. J. Wood (Eds.), *Thermodynamics of minerals and melts* (Advances in Physical Geochemistry Vol. 1, pp. 19–34). Springer. https://doi.org/10.1007/978-1-4612-5871-1_2

Holland, T., & Powell, R. (1985). An internally consistent thermodynamic dataset with uncertainties and correlations: 2. Data and results.

Journal of Metamorphic Geology, 3, 343–370. https://doi.org/10.1111/j.1525-1314.1985.tb00325.x

Holland, T., & Powell, R. (1990). An enlarged and updated internally consistent thermodynamic dataset with uncertainties and correlations: The system $K_2ONa_2OCaOMgOMnOFeOFe_2O_3Al_2O_3TiO_2SiO_2CH_2O_2$. *Journal of Metamorphic Geology, 8*(1), 89–124. https://doi.org/10.1111/j.1525-1314.1990.tb00458.x

Holland, T., & Powell, R. (1991). 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*, 265–273. https://doi.org/10.1007/BF00306484

Holland, T., & Powell, R. (1992). Plagioclase feldspars: Activity-composition relations based upon Darken's quadratic formalism and Landau theory. *American Mineralogist, 77*, 53–61.

Holland, T., & Powell, R. (1996a). Thermodynamics of order-disorder in minerals; I. Symmetric formalism applied to minerals of fixed composition. *American Mineralogist, 81*(11-12), 1413–1424. https://doi.org/10.2138/am-1996-11-1214

Holland, T., & Powell, R. (1996b). Thermodynamics of order-disorder in minerals; II. Symmetric formalism applied to solid solutions. *American Mineralogist, 81*(11-12), 1425–1437. https://doi.org/10.2138/am-1996-11-1215

Holland, T., & Powell, R. (1998). An internally consistent thermodynamic data set for phases of petrological interest. *Journal of Metamorphic Geology, 16*(3), 309–343. https://doi.org/10.1111/j.1525-1314.1998.00140.x

Holland, T., & Powell, R. (2003). Activity–composition relations for phases in petrological calculations: an asymmetric multicomponent formulation. *Contributions to Mineralogy and Petrology, 145*, 492–501. https://doi.org/10.1007/s00410-003-0464-z

Holland, T., & Powell, R. (2006). Mineral activity-composition relations and petrological calculations involving cation equipartition in multisite minerals: a logical inconsistency *Journal of Metamorphic Geology, 24*(9), 851–861. https://doi.org/10.1111/j.1525-1314.2006.00672.x

Holland, T., & Powell, R. (2011). 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. https://doi.org/10.1111/j.1525-1314.2010.00923.x

Huang, Y. K.. & Chow, C. Y. (1974). The generalized compressibility equation of Tait for dense matter. *Journal of Physics D: Applied Physics, 7*(15), 2021–2023. https://doi.org/10.1088/0022-3727/7/15/305

Kelly, E. D., Carlson, W. D., & Ketcham, R. A. (2013). Magnitudes of departures from equilibrium during regional metamorphism of porphyroblastic rocks. *Journal of Metamorphic Geology, 31*(9), 981–1002. https://doi.org/10.1111/jmg.12053

Kohn, M. J., & Spear, F. S. (1991a). Error propagation for barometers: 2. Application to rocks. *American Mineralogist, 76*, 138–147.

Kohn, M. J., & Spear, F. S. (1991b). Error propagation for barometers: 1. Accuracy and precision of experimentally located end-member reactions. *American Mineralogist, 76*, 128–137.

Korzhinskii, D. S. (1959). *Physicochemical basis of the analysis of the paragenesis of minerals* (Fizikokhimicheskie osnovy analiza paragenezisov mineralov). Consultants Bureau.

Krumbein, W. C. (1962). The computer in geology. *Science, New Series, 136*, 1087–1092. https://doi.org/10.1126/science.136.3522.1087

Krupka, K. M., Robie, R. A., & Hemingway, B. S. (1979). High temperature heat capacities of corundum, periclase, anorthite, $CaAl_2Si_2O_8$ glass, muscovite, pyrophyllite, $KAlSi_3O_8$ glass, grossular, and $NaAlSi_3O_8$ glass. *American Mineralogist, 64*, 86–101.

Labotka, T. C. (1981). Petrology of an andalusite-type region metamorphic terrain, Panamint Mountains, California. *Journal of Petrology, 22*(2), 261– 296. https://doi.org/10.1093/petrology/22.2.261

Lanari, P., & Duesterhoeft, E. (2019). Modeling metamorphic rocks using equilibrium thermodynamics and internally consistent databases: Past achievements, problems and perspectives. *Journal of Petrology, 60*(1), 19–56. https://doi.org/10.1093/petrology/egy105

Lanari, P., & Engi, M. (2017). Local Bulk Composition Effects on Metamorphic Mineral Assemblages. In M. J. Kohn, M. Engi, & P. Lanari (Eds.), *Petrochronology: Methods and Applications* (RiMG Vol. 83, pp. 55–102). Mineralogical Society of America. https://doi.org/10.2138/rmg.2017.83.3

Lange, R. (1997). A revised model for the density and thermal expansivity of $K_2ONa_2OCaOMgOAl_2O_3SiO_2$ liquids from 700 to 1900 K: extension to crustal magmatic temperatures. *Contributions to Mineralogy and Petrology, 130*, 1–11. https://doi.org/10.1007/s004100050345

Liu, Q., Proust, C., Gomez, F., Luart, D., & Len, C. (2020). The prediction multi-phase, multi reactant equilibria by minimizing the gibbs energy of the system: Review of available techniques and proposal of a new method based on a Monte Carlo technique. *Chemical Engineering Science, 216*, 1–16. https://doi.org/10.1016/j.ces.2019.115433

Macdonald, J. R. (1966). Some simple isothermal equations of state. *Reviews of Modern Physics, 38*(4), 669–679. https://doi.org/10.1103/RevModPhys.38.669

Marmo, B. A., Clarke, G. L., & Powell, R. (2002). Fractionation of bulk rock composition due to porphyroblast growth; effects on eclogite facies mineral equilibria, Pam Peninsula, New Caledonia. *Journal of Metamorphic Geology, 20*(1), 151–165. https://doi.org/10.1046/j.0263-4929.2001.00346.x

Mesmer, R. E., Marshall, W. L., Palmer, D. A., Simonson, J. M., & Holmes, H. F. (1988). Thermodynamics of aqueous association and ionization reactions at high temperatures and pressures. *Journal of Solution Chemistry, 17*(8), 699–718. https://doi.org/10.1007/BF00647417

Michelsen, M. L. (1982). The isothermal flash problem. Part I. Stability. *Fluid Phase Equilibria, 9*(1), 1–19. https://doi.org/10.1016/0378-3812(82)85001-2

Mikhail, E. M. (1976). *Observations and least squares*. Dun-Donnelley.

Newton, R. C. (1983). Geobarometry of high-grade metamorphic rocks. *American Journal of Science, 283-A*, 1–28.

Newton, R. C., & Haselton, H. T. (1981). Thermodynamics of the garnet-plagioclase-Al_2Si0_5-quartz geobarometer. In R. C. Newton, A. Navrotsky, & B. J. Wood (Eds.), *Thermodynamics of minerals and melts* (Advances in Physical Geochemistry Vol. 1, pp. 131–147). Springer. https://doi.org/10.1007/978-1-4612-5871-1_7

Oreskes, N. (1998). Evaluation (not validation) of quantitative models. *Environmental Health Perspectives Supplements, 106*, 1453–1460. https://doi.org/10.1289/ehp.98106s61453

Oreskes, N., Shrader-Frechette, K., & Belitz, K. (1994). Verification, validation, and confirmation of numerical models in the Earth sciences. *Science, 263*, 641–646. https://doi.org/10.1126/science.263.5147.641

Palin, R. M., Weller, O. M., Waters, D. J., & Dyck, B. (2016). Quantifying geological uncertainty in metamorphic phase equilibria modelling: A monte carlo assessment and implications for tectonic interpretations.

Geoscience Frontiers, 7(4), 591–607. https://doi.org/10.1016/j.gsf.2015.08.005

Pitzer, K. S., & Sterner, S. M. (1994). Equations of state valid continuously from zero to extreme pressures for H_2O and CO_2. *The Journal of Chemical Physics, 101*(4), 3111-3116. https://doi.org/10.1063/1.467624

Powell, R. (1978). *Equilibrium thermodynamics in petrology*. Harper & Row.

Powell, R. (1985). Geothermometry and geobarometry - a discussion. *Journal of the Geological Society of London, 142*(1), 29–38. https://doi.org/10.1144/gsjgs.142.1.0029

Powell, R. (1987). Darken's quadratic formalism and the thermodynamics of minerals. *American Mineralogist, 72*, 1–11.

Powell, R., & Downes, J. (1990). Garnet porphyroblast-bearing leucosomes in metapelites: Mechanisms, phase diagrams, and an example from Broken Hill, Australia. In J. R. Ashworth & M. Brown (Eds.), *High-temperature metamorphism and crustal anatexis* (The Mineralogical Society Series Vol. 2, pp. 105–123). Kluwer Academic Publishers. https://doi.org/10.1007/978-94-015-3929-6_5

Powell, R., & Holland, T. (1985). An internally consistent thermodynamic dataset with uncertainties and correlations: 1. Methods and a worked example. *Journal of Metamorphic Geology, 3*(4), 327–342. https://doi.org/10.1111/j.1525-1314.1985.tb00324.x

Powell, R., & Holland, T. (1988). An internally consistent dataset with uncertainties and correlations: 3. Applications to geobarometry, worked examples and a computer program. *Journal of Metamorphic Geology, 6*(2), 173–204. https://doi.org/10.1111/j.1525-1314.1988.tb00415.x

Powell, R., & Holland, T. (1990). Calculated mineral equilibria in the pelite system, KFMASH ($K_2OFeOMgOAl_2O_3SiO_2H_2O$). *American Mineralogist, 75*, 367–380.

Powell, R., & Holland, T. (1993). The applicability of least squares in the extraction of thermodynamic data from experimentally bracketed mineral equilibria. *American Mineralogist, 78*, 107–112.

Powell, R., & Holland, T. (1994). Optimal geothermometry and geobarometry. *American Mineralogist, 79*, 120–133.

Powell, R., & Holland, T. (2008). On thermobarometry. *Journal of Metamorphic Geology, 26*(2), 155–179. https://doi.org/10.1111/j.1525-1314.2007.00756.x

Powell, R., Holland, T., & Worley, B. (1998). Calculating phase diagrams involving solid solutions via non-linear equations, with examples using thermocalc. *Journal of Metamorphic Geology, 16*(4), 577–588. https://doi.org/10.1111/j.1525-1314.1998.00157.x

Putirka, K. D. (2008). Thermometers and barometers for volcanic systems In K. D. Putirka & F. J. Tepley (Eds.), *Minerals, Inclusions and Volcanic Processes* (RiMG Vol. 69, pp. 61–120). Mineralogical Society of America. https://doi.org/10.2138/rmg.2008.69.3

Reid, R. C., Prausnitz, J. M., & Sherwood, T. K. (1987). *The properties of gases and liquids: Their estimation and correlation* (3rd ed.). Mcgraw-Hill Book Company.

Reverdatto, V. V., Likhanov, I. I., Polyansky, O. P., Sheplev, V. S., & Kolobov, V. Y. (2019). Mineral Geothermobarometry. In *The Nature and Models of Metamorphism* (pp. 55–82). Springer. https://doi.org/10.1007/978-3-030-03029-2_2

Riel, N., Kaus, B. J. P., Green, E. C. R., & Berlie, N. (2022). Magemin, an efficient Gibbs energy minimizer: Application to igneous systems. *Geochemistry, Geophysics, Geosystems, 23*(7), 1–27. https://doi.org/10.1029/2022GC010427

Robie, R., Hemingway, B. S., & Fishert, J. R. (1978). *Thermodynamic properties of minerals and related substances at 298.15 K and 1 bar (10^5 pascals) pressure and at higher temperatures* (U.S. Geological Survey Bulletin 1452). U.S. Government Printing Office. https://pubs.er.usgs.gov/publication/b1452

Robie, R. A., & Hemingway, B. S. (1995). *Thermodynamic properties of minerals and related substances at 298.15 K and 1 bar (10^5 pascals) pressure and at higher temperatures* (U.S. Geological Survey Bulletin, 2131, pp. 1–461). U.S. Geological Survey. https://pubs.usgs.gov/bul/2131/report.pdf

Roddick, J. C. (1987). Generalized numerical error analysis with applications to geochronology and thermodynamics. *Geochimica et Cosmochimica Acta, 51*(8), 2129–2135. https://doi.org/10.1016/0016-7037(87)90261-4

Rossum, G. V., Warsaw, B., & Coghlan, N. (2001). *PEP 8 Style guide for python code. https://peps.python.org/pep-0008/ (accesed on January, 2023)*. Python.org.

Saxena, S. K. (1969). Silicate solid solutions and geothermometry 1. Use of the regular solution model. *Contributions to Mineralogy and Petrology, 21*, 338–345. https://doi.org/10.1007/BF02672805

Smith, W. R., & Missen, R. W. (1982). *Chemical reaction equilibrium analysis: Theory and Algorithms*. John Wiley & Sons.

Spear, F. S. (1989). Relative thermobarometry and metamorphic P-T paths. In J. S. Daly, R. A. Cliff, & B. W. D. Yardley (Eds.), *Evolution of Metamorphic Belts* (GSL Special Publications Vol. 43, pages 1–44). Geological Society of London. https://doi.org/10.1144/GSL.SP.1989.043.01.04

Spear, F. S. (2017). Garnet growth after overstepping. *Chemical Geology, 466*, 491–499. https://doi.org/10.1016/j.chemgeo.2017.06.038

Spear, F. S., Ferry, J. M., & Rumble, D. (1982). Analytical formulation of phase equilibria: the Gibbs method. In J. M. Ferry (Ed.), *Characterization of Metamorphism through Mineral Equilibria* (Reviews in Mineralogy Vol. 10, pp. 105–152). Mineralogical Society of America. https://doi.org/10.1515/9781501508172-008

Spear, F. S., & Pattison, D. R. M. (2017). The implications of overstepping for metamorphic assemblage diagrams (MADs). *Chemical Geology, 457*, 38–46. https://doi.org/10.1016/j.chemgeo.2017.03.011

Spear, F. S., & Wolfe, O. M. (2018). Evaluation of the effective bulk composition (EBC) during growth of garnet. *Chemical Geology, 491*, 39–47. https://doi.org/10.1016/j.chemgeo.2018.05.019

Stuwe, K. (1997). Effective bulk composition changes due to cooling: a model predicting complexities in retrograde reaction textures. *Contributions to Mineralogy and Petrology, 129*, 43–52. https://doi.org/10.1007/s004100050322

Tinkham, D. K., & Ghent, E. D. (2005). Estimating p-t conditions of garnet growth with isochemical phase-diagram sections and the problem of effective bulk-composition. *The Canadian Mineralogist, 43*(1), 35–50. https://doi.org/10.2113/gscanmin.43.1.35

Van Ness, H. C. (1983). *Understanding thermodynamics*. Dover Publications.

Wei, C. J., & Powell, R. (2003). Phase relations in high-pressure metapelites in the system KFMASH (K_2O–FeO–MgO–Al_2O_3–SiO_2–H_2O) with application to natural rocks. *Contributions to Mineralogy and Petrology, 145*, 301–315. http://dx.doi.org/10.1007/s00410-003-0454-1

Wei, C. J., & Powell, R. (2006). Calculated phase relations in the system NCKFMASH (Na_2O–CaO–K_2O–FeO–MgO–Al_2O_3–SiO_2–H_2O) for high-pressure metapelites. *Journal of Petrology, 47*, 385–408. https://doi.org/10.1093/petrology/egi079

Wei, C. J., Powell, R., & Clarke, G. L. (2004). Calculated phase equilibria for low- and medium-pressure metapelites in the KFMASH and KMnFMASH systems. *Journal of Metamorphic Geology, 22*(5), 495–508. https://doi.org/10.1111/j.1525-1314.2004.00530.x

White, R. W., Powell, R., Holland, T., Johnson, T. E., & Green, E. C. R. (2014). New mineral activity–composition relations for thermodynamic calculations in metapelitic systems. *Journal of Metamorphic Geology, 32*(3), 261–286. https://doi.org/10.1111/jmg.12071

Will, T. E., & Powell, R. (1992). Activity-composition relationships in multicomponent amphiboles: An application of Darken's quadratic formalism. *American Mineralogist, 77*, 954–966.

Williams, E. J. (1935). The effect of thermal agitation on atomic arrangement in alloys III. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 152*, 231–252. https://doi.org/10.1098/rspa.1934.0132

Worley, B., & Powell, R. (2000). High-precision relative thermobarometry: theory and a worked example. *Journal of Metamorphic Geology, 18*(1), 91–101. https://doi.org/10.1046/j.1525-1314.2000.00239.x

Xiang, H., & Connolly, J. A. D. (2022). GeoPS: An interactive visual computing tool for thermodynamic modelling of phase equilibria. *Journal of Metamorphic Geology, 40*(2), 243–255. https://doi.org/10.1111/jmg.12626

Ziberna, L., Green, E. C. R., & Blundy, J. D. (2017). Multiple-reaction geobarometry for olivine-bearing igneous rocks. *American Mineralogist, 102*, 2349–2366. https://doi.org/10.2138/am-2017-6154

Zuluaga, C. A., Stowell, H. H., & Tinkham, D. K. (2005). The effect of zoned garnet on metapelite pseudosection topology and calculated metamorphic P-T paths. *American Mineralogist, 90*, 1619–1628. https://doi.org/10.2138/am.2005.1741

Index