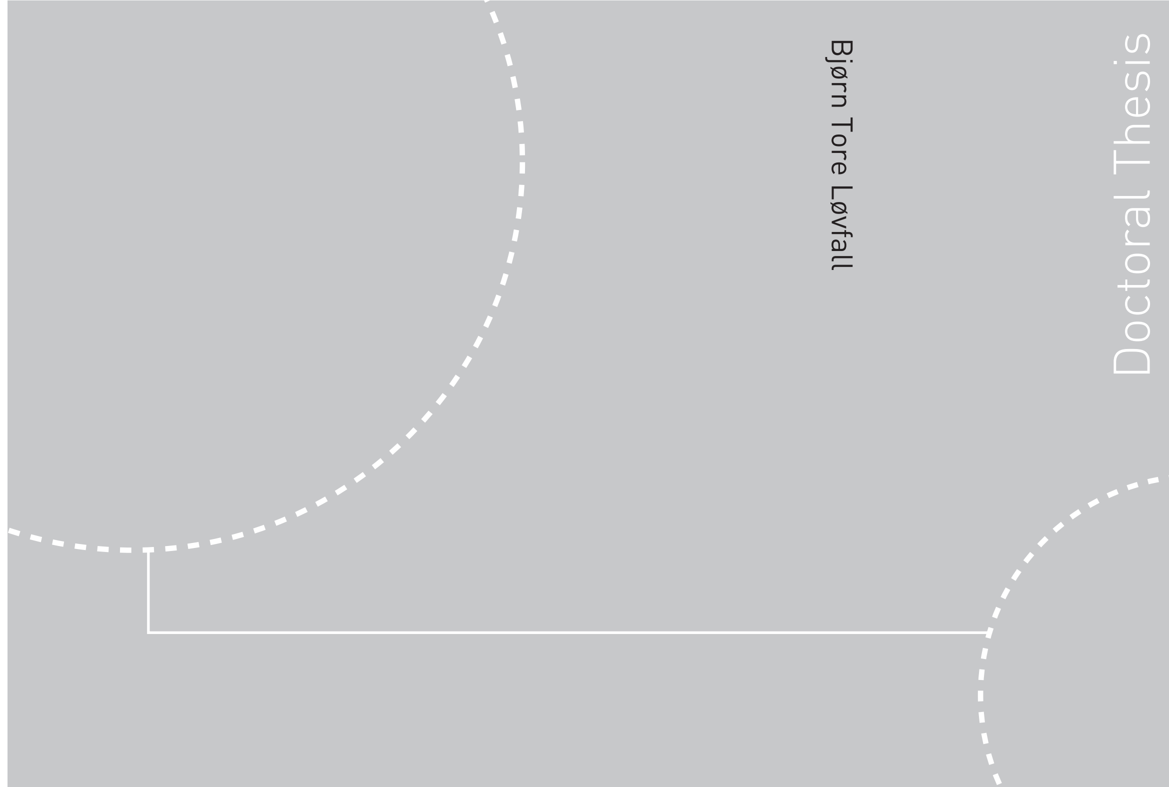
Doctoral theses at NTNU, 2008:323

Bjørn Tore Løvfall

Computer Realization of Thermodynamic Models Using Algebraic Objects



ISBN 978-82-471-1355-4 (printed ver.)
ISBN 978-82-471-1356-1 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2008:323

NTNU
Norwegian University of
Science and Technology
Thesis for the degree of
philosophiae doctor
Faculty of Natural Sciences and Technology
Department of Chemical Engineering

NTNU

Bjørn Tore Løvfall

Computer Realization of Thermodynamic Models Using Algebraic Objects

Thesis for the degree of philosophiae doctor

Trondheim, December 2008

Norwegian University of
Science and Technology
Faculty of Natural Sciences and Technology
Department of Chemical Engineering

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of philosophiae doctor

Faculty of Natural Sciences and Technology
Department of Chemical Engineering

©Bjørn Tore Løvfall

ISBN 978-82-471-1355-4 (printed ver.)
ISBN 978-82-471-1356-1 (electronic ver.)
ISSN 1503-8181

Doctoral Theses at NTNU, 323

Printed by Tapir Uttrykk

Abstract

The main focus of this thesis is to make thermodynamic models available for different computer programs on Windows, Linux and Unix computer platforms. In thermodynamic calculations, like phase or reaction equilibrium, the first and maybe higher order gradients of the thermodynamic functions are needed. The major challenge has been to enable higher order gradients (> 2) of such functions, with symbolic precision, and without loosing the structure of the thermodynamic model equations. The solution to the problem lies in the novel RGrad language which enables commutative multidimensional array operations. The RGrad language introduced in this thesis is algebraically closed with respect to differentiation, which means that gradient calculations of arbitrary order are possible. The results are translated into ANSI C-code, interfaced with the user code by using a small and well defined C-interface. This interface has been demonstrated for Ruby and Matlab, but can easily be extended to other languages. The RGrad language has also been used to provide automatically generated Legendre transforms, with gradients.

A few thermodynamic models generated with this methodology have been used to develop a new method for checking phase stabilities in multiphase, near-critical, phase equilibria. The method makes use of Legendre transforms to formulate the problem such that the same equations can be used for any number of phases, with any number of components. The phase stability test traverses the extensive thermodynamic space in the search for the most stable phase while keeping the chemical potentials the same as in the test phase.

The final application discussed in this thesis deals with critical and tri-critical points. Legendre transforms are used to formulate stability criteria which are the same regardless of the number of components in the system. Furthermore, a Taylor expansion at the critical point has been developed to approximate the two-phase boundary of a system. The success of these calculations demonstrates that the gradient calculations are correct, and also that the automatic Legendre transform is implemented correctly.

Acknowledgments

Four and a half years of hard work has come to an end. This would not have been possible without the help of others. During the work with this thesis, I have received help and support from many people. First of all I would like to thank Associate professor Tore Haug-Warberg, my supervisor, for more than five years of cooperation. His suggestions and insight has made this work possible. I would also like to extend my appreciation to Professor Heinz A. Preisig, my co-supervisor, for many fruitful discussions. The project has been funded by the Norwegian Research Council through the ThermoTech KMB program.

Further I would like to thank my former and present colleagues at the process system engineering group at the Department of Chemical Engineering. I am grateful to all of you for making my stay a memorable one. I would especially like to thank Olaf who always encouraged me to write a better code. I would also thank him for his many good suggestions. I want to express my gratitude to Ivan for reading parts of my thesis, and to Jørgen, for many inspiring conversations during the three years we shared office.

Finally I would like to thank my friends and my supportive family.

Contents

Abstract iii

Acknowledgements v

List of Symbols ix

1 Introduction 1
1.1 Motivation 2
1.2 Thesis Overview 3

I Thermodynamics 5

2 Theory 7
2.1 The Legendre Transform 7
2.2 Stability 17

3 Phase Stability Test 23
3.1 Example from Literature 24
3.2 Alternative Energy Function Formulation 28
3.3 An Additional Example 38
3.4 The Use of ODE Solvers 39
3.5 Using DAE Solvers in Matlab 45
3.6 Conclusion 45

4 Criticality 49
4.1 Critical Lines in Binary Systems 50
4.2 The Tri-critical Point 56
4.3 Taylor Expansion of Phase Boundaries 58
4.4 Conclusion 62

II The RGrad Language **63**

5 Background **65**
5.1 Automatic Differentiation 65
5.2 Portable Thermodynamics Software 71
5.3 Remarks 73

6 Language Definition **75**
6.1 Introduction 76
6.2 Grammar Definition 81
6.3 Gradient Calculations 84
6.4 Broadcasting 86

7 Model Generation **91**
7.1 Unit Consistency Check 92
7.2 User Interface 93
7.3 Internal Representation 102
7.4 Automatic Gradient Calculations 103
7.5 Automatic Legendre Transforms 107
7.6 Potential Pitfalls 111

8 Code Generation **117**
8.1 C-Code 118
8.2 C-API 127
8.3 Ruby Interface 129
8.4 Matlab mex Interface 132
8.5 Additional Features 135
8.6 Library Implementation 138
8.7 Testing and Validation 149

9 Concluding Remarks **151**
9.1 Recommendations and Further Work 153

Bibliography **155**

A Higher Order Gradients of the Legendre Transform **161**

B SRK Model Parameters **167**

C Code Examples **169**

List of Symbols

α	Dimensionless variable defined as $\alpha = \frac{\lvert T_\circ \rvert S}{\lvert T_\circ \rvert S + \lvert p_\circ \rvert V}$
δ	Kronecker delta
$\mathbf{n}$	Set of mole numbers
μ	Set of chemical potentials
μ_i	Chemical potential of component i
Ω	Grand canonical potential
σ	Legendre transformed variable $\sigma = \frac{\partial f}{\partial r}$
θ	Same as t
$\tilde{X}$	Unscaled tangent plane distance in X.
$\tilde{x}$	Scaled tangent plane distance in X.
ξ_i	Used in Legendre transforms to give transformed variables
A	Helmholtz energy
C	Constant defined as $\lvert T_\circ \rvert S + \lvert p_\circ \rvert V$
G	Gibbs energy
H	Enthalpy
N_i	Mole numbers of component i
p	Pressure
$p_\circ$	Pressure at a reference state, also used for other variables.
r	Variable that will be Legendre transformed

S Entropy

T Temperature

t Variable that will not be Legendre transformed

U Internal energy

V Volume

X Unnamed thermodynamic potential in variables S, V, μ_i

x_i Used in Legendre transforms to give untransformed variables

$y_{kk}^{(k-1)}$ The k-1 Legendre transform of $y^{(0)}$ differentiated twice with respect to the kth variable.

Chapter 1

Introduction

Solving thermodynamic equilibrium problems using gradient methods creates a fundamental need for derivatives. Traditionally, the derivatives are derived manually and hand coded, usually limited to the gradient or occasionally including the Hessian, or, alternatively, they are calculated by numerical differentiation. Hand-coded expressions yield a highly optimized code, but are expensive and error prone to implement. Numerical differentiation is easy to implement, but requires more computational effort, and the results are less accurate.

Several approaches to provide symbolic derivatives of thermodynamic functions have been presented in the literature. The approach by Thiéry (1996) enables up to the fourth derivative of multicomponent thermodynamic functions. This is achieved by recognizing common expression patterns in the functions, and implementing special derivative routines for these. Taylor (1997, 1998a,b) uses the computer algebra system Maple to calculate the derivatives of thermodynamic functions. This enables higher order derivatives, but the results are restrained by the Maple software. The benefits of analytical derivatives in dynamic simulation is demonstrated by Kilakos and Kalitventzeff (1993). They have developed an algorithm for calculating the derivatives needed based on a small set of hand-coded partial derivatives.

Despite the obvious advantage of using symbolic gradients very little activity is registered in this area. This work intends to fill a gap in the existing methodologies for automatic gradient calculations.

1.1 Motivation

In order to calculate the gradient symbolically, some kind of structure in the formulation is needed. If not, all operations will be scalar, and a computer algebra system, like Maple of Mathematica, can easily be used. Our observation is that a typical thermodynamic expression involves an inner product between an interaction parameter matrix and a mole number vector. This makes computer algebra systems less desirable for calculating the derivatives, because these programs focus on creating human-readable results from scalar differentiations. Another approach for calculating accurate derivatives is automatic differentiation, but the inherent structure of the thermodynamic expressions is very difficult to maintain by this approach, as all expressions are first broken down to scalar operations and then differentiated. The motivation for this work is to develop a novel method for automatic gradient calculation of thermodynamic models to higher order (> 2), without destroying the algebraic structure of the model.

The use of thermodynamic models are omnipresent in the chemical process industry. Thermodynamic models are being used in a variety of applications, such as flow sheet simulations, spreadsheet calculations and even in advanced process control. In industrial applications it is important that the models produce the same numbers in the entire organization, quite independent of the implementation of the code and what machine is being used. To a large extent this can be obtained by using large proprietary program packages. The problem with these packages is that they are made with the intention of being able to serve everyone's needs. This makes them large and complex, both to use and to maintain. The approach taken here is quite the opposite; instead of creating a generic tool for instantiating a general purpose thermodynamic engine, a tool for creating small and self-contained portable models is attempted. This enables the user to solve specific problems with a high degree of confidence, and low cost.

Since most programs today are able to incorporate external program modules, usually through a dynamically linked library, the goal here is to produce self-contained ANSI C-code that is easy to integrate with other programs. Self-contained means that the final code does not rely on any external resources. Consequently no software update will make the code unusable in the future, no licence server will stop the code from running, and no database update will make the code produce different results from one day to the next. The change of computer platform will, within the machine precision, reproduce the same numerical results. All data used for calculating the gradients are contained within the model, and the user can both read and change the parameters and variables according to his needs.

1.2 Thesis Overview

This work focus on two different topics and it was decided to divide the thesis into two distinct parts: Part I with focus on thermodynamic calculations, and Part II with focus on code generation.

Part I: Chapter 2 briefly reviews the thermodynamic background used in the remaining chapters of Part I, where the main focus is put on the Legendre transform, and on thermodynamic stability calculations. The Legendre transform is also used in Part II, where the automatic Legendre transform is included as a features of the language defined there. Chapter 3 and 4 present calculations concerning phase stability, critical and tri-critical point calculations. These chapters requires thermodynamic models with the features as the ones created with the tool described in Part II. The chapters of Part I are therefore also meant as motivation for Part II. Finally, the calculations made in Part I are meant to demonstrate that the models created by the tool in Part II are giving probable results.

Part II: Consists of four chapters, where Chapter 5 is giving some background for the rest of the part. In Chapter 6 a language for defining differentiable models with inner product structure is presented, and a technique for enabling multidimensional array operations, called broadcasting, is introduced. This language is later used to construct an input language which is documented in Chapter 7. Chapter 8 documents the different outputs that can be generated. Chapter 9 concludes the work, and gives suggestions for further research.

Part I

Thermodynamics

Chapter 2

Theory

This chapter covers the theory used in Chapters 3 and 4. The Legendre transform is needed in order to define energy potentials with state variables different from the ones used in the underlying thermodynamic model, while the stability analysis is used to locate stable phases in a heterogeneous equilibrium mixture. (Tri-)critical points are also discussed.

2.1 The Legendre Transform

Thermodynamic calculations often utilise several sets of independent variables, whereas the required number of independent variables for a system is given by the system's intensive degrees of freedom, which according to Gibbs phase rule is:

$$F = C - P + 2 \tag{2.1}$$

Thus, the degrees of freedom (F) are determined by the number of components (C) and the number of phases (P). From Equation 2.1 single-phase systems have the maximum degrees of freedom since the introduction of a new phase will impose a constraint to the system. For example in a single component system with one phase the phase rule states two degrees of freedom. This means that two intensive variables are needed to describe the system, for instance temperature (T) and pressure (p), or pressure and chemical potential (μ). Since these are all intensive variables they do not tell anything about the size of the system, thus an additional (extensive) variable must be added if this information is needed.

It is clearly possible to choose different combinations of independent variables, and the particular choice is often based on tradition rather than convenience. The

typical engineer does not ask himself "what is the natural choice of independent variables in my system?" He would probably measure the temperature and pressure, and in simulations he would use a Gibbs energy model, for which T and p are the canonical variables. The canonical variables are the set of variables that define the fundamental relations which contain all possible thermodynamic-related information about the system, see Callen (1985). These variables are also referred to as the natural variables, as discussed by Alberty (1994). By that he defines the variables which are natural for describing the system. For instance in the equation below:

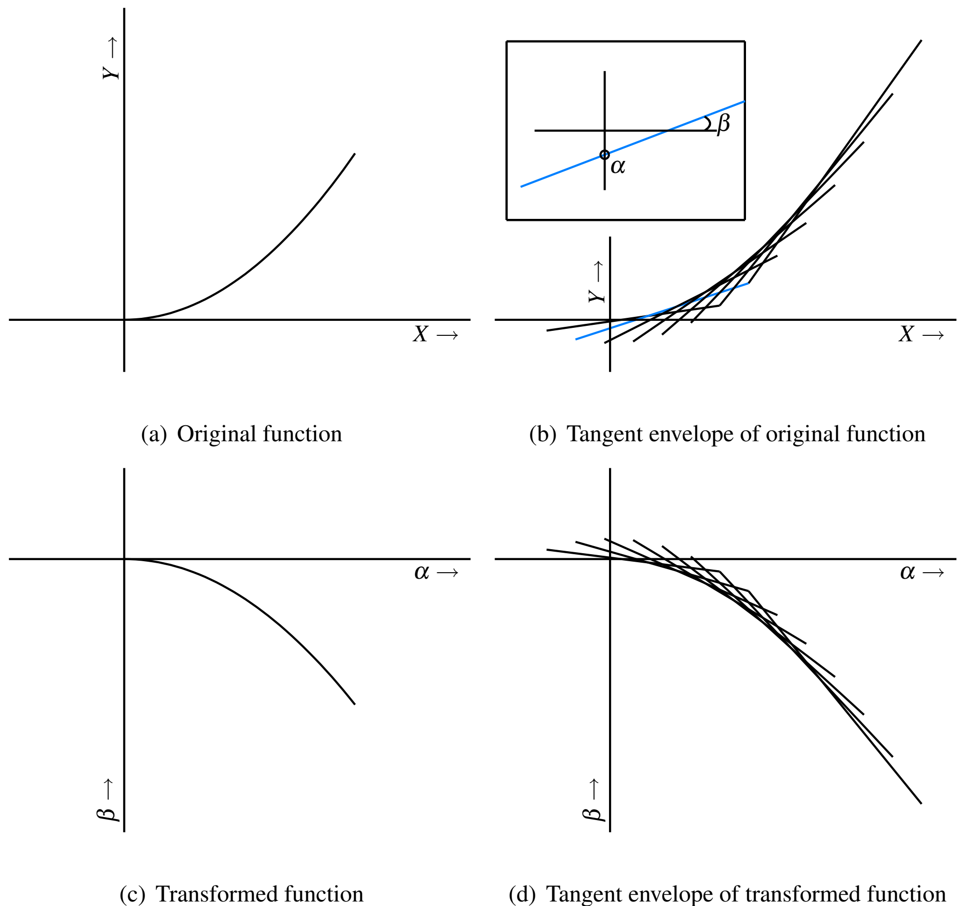
$$\mathrm{d}U = T\,\mathrm{d}S - p\,\mathrm{d}V + \sum_{i=1}^{n} \mu_i\,\mathrm{d}N_i \tag{2.2}$$

If U is determined as a function of S, V and N_i, the quantities T, p and μ_i can be obtained as partial derivatives. S,V,N_i are therefore natural variables of U. Note that in Equation 2.2 (S, T), (V, p) and (N_i, μ_i) appear as conjugate variable pairs. A pair consists of an extensive and its conjugate intensive variable. S, V and N_i are extensive, while T, p and μ_i are intensive.

The current practice it to represent the thermodynamic information as a function of the observed properties, such as temperature, volume or pressure and so forth. But for the particular application it may be more appropriate to use another variable space for the representation of the system's state. In thermodynamics this can be achieved by transforming to a different set of independent variables with the help of Legendre transforms, as shown by Alberty (1994); Brendsdal (1999); Haug-Warberg (2006).

It is sometimes convenient to replace some of the extensive variables in U with their conjugate intensive variables. This will in principle be the same as replacing the variable by its partial derivative. According to Callen (1985), thermodynamic functions are logically complete and self contained within both the entropy and the energy representation. This shows it is possible to use the Legendre transform for mapping into an alternative variable space, a method which is conveniently visualised geometrically. As an example, assume an arbitrary function with one independent variable $y = y(x)$. Assume furthermore that y is continuous and at least twice differentiable with respect to the independent variable. The function can be then reformulated to $y = y(\frac{\partial y}{\partial x}) = y(s)$. It is important that this transform is performed such that the information content of the original function is retained, because only then can the transformation be inverted. At first sight, the inversion seems possible by solving the respective differential equation, but by doing so one observes that while the shape of the original function is retained, the integration

constant is unknown. Some information from the original function will therefore be lost by doing so. This problem is magically avoided by using the Legendre transforms.



(a) Original function (b) Tangent envelope of original function

(c) Transformed function (d) Tangent envelope of transformed function

Figure 2.1: Graphical representation of the Legendre transform. Figure (a) and (b) are used to create Figure (c). This, in combination with Figure (d) are used to recreate Figure (a)

Figure 2.1 shows a graphical representation of the Legendre transform. Figure 2.1(a) displays the original function, while Figure 2.1(b) is constructed by plotting the tangents to selected points of the first figure. The slopes of the tangents (β), and the intersection with the y-axis (α), are used to construct Figure 2.1(c). Figure 2.1(d) is shown for completeness. In this case the tangent lines are constructed from Figure 2.1(c) can be used to reconstruct Figure 2.1(a).

The Legendre transform exploits the basic principle that a curve, originating from the graph of a function, can be described as a locus of points constructed from the tangent in each point. In other words: A line in Cartesian coordinates can be described by pairs of x and y coordinates. These pairs make up the graph in the coordinate system. For each set (x,y), there is another set (α,β), which can be used to represent all the tangent lines that constructs the envelope from which the original function is defined. Here, β represents the slope, and α the point where the tangent intersects the y-axis. α and β can then be used to construct a new graph (function). If the same methodology is applied to the new graph the original graph can be retained.

There are a few exceptions to this, as mentioned above, namely that the function must be continuous, and twice differentiable. If the function is non-differentiable, then it is impossible to construct the tangent, and thereby impossible to map the function into the new coordinates. In order to get back to the original function, the second derivative of the original function with respect to the transformed variable is required, and this cannot be a constant value. If the second order differential is not defined, information would be lost by changing coordinates. Another problem with transforming back to the original function occurs if the original function has tangents which intersect the graph at some point. This means that two different values of (x,y) give the same values of (α, β). It is then impossible to recalculate both values in the reverse transform, and therefore information is lost. For these reasons the function must be twice differentiable, and the second derivative must not change sign.

The Legendre transform was demonstrated here for a function with one variable. An extension to several variables is straightforward, but the graphical representation is of course more complicated. The general function

$$y^{(0)}(x_1, x_2, \ldots, x_n) \tag{2.3}$$

can be transformed with respect to all the x_n variables in any combination. In order to simplify the transform, a new variable is often introduced:

$$\xi_i \triangleq \left(\frac{\partial y^{(0)}}{\partial x_i}\right)_{x_1,\ldots,[x_i],\ldots x_n} \tag{2.4}$$

Here $[x_i]$ means that x_i is not held constant. A new function can now be defined, based on 2.3:

$$y^{(1)}(\xi_1, x_s, \ldots, x_n) = y^{(0)} - \xi_1 x_1 . \tag{2.5}$$

The variables of y are ordered such that the transformed variable always appears first. Equation 2.5 can be generalised to,

$$y^{(k)}(\xi_1, \xi_2, \ldots, \xi_k, x_{k+1}, \ldots, x_n) = y^{(0)} - \sum_{i=1}^{k} \xi_i x_i \tag{2.6}$$

Here, superscript (k) represents the order of the transform, and by setting $k = 1$ this reduces to Equation 2.5. By using the thermodynamic potential $U(S, V, N_1, \ldots, N_m)$ as a starting point other well known thermodynamic potentials such as Gibbs energy and Helmholtz energy can be defined (Beegle et al., 1974a).

When a thermodynamic potential is determined as a function of its canonical variables, all other thermodynamic properties of the system can be calculated (Alberty, 1994). Equation 2.7 shows an example of how the Legendre transform is used to transform from internal energy into Helmholtz energy, and back again.

Starting out with

$$U^{(0)} = U(S, V, \mathbf{n}), \tag{2.7a}$$

and applying Equation 2.5 we get Helmholtz energy:

$$U^{(1)} = U^{(0)} - \frac{\partial U^{(0)}}{\partial S} S \triangleq A^{(0)} = A(T, V, \mathbf{n}), \tag{2.7b}$$

Thereby using the definition:

$$\frac{\partial U^{(0)}}{\partial S} \triangleq T. \tag{2.7c}$$

Again, by using Equation 2.5 the inverse transform is defined as:

$$A^{(1)} = A^{(0)} - \frac{\partial A^{(0)}}{\partial T} T, \tag{2.7d}$$

$$\frac{\partial A^{(0)}}{\partial T} = \frac{\partial U^{(0)}}{\partial T} - S - T \frac{\partial S}{\partial T} = -S, \tag{2.7e}$$

$$\frac{\partial U^{(0)}}{\partial T} = 0, \tag{2.7f}$$

$$\frac{\partial S}{\partial T} = 0, \tag{2.7g}$$

This proves the desired result:

$$A^{(1)} = U^{(0)} - TS - T(-S) = U^{(0)}. \tag{2.7h}$$

Table 2.1: Table of transforms

Potential	Natural variables	Conjugate variables	Legendre transform
U	$S,V,\mathbf{n}$	$T,-p,\mu$	U
A	$T,V,\mathbf{n}$	$-S,-p,\mu$	$U-TS$
H	$S,-p,\mathbf{n}$	$T,-V,\mu$	$U+pV$
X	S,V,μ	$T,-p,-\mathbf{n}$	$U-\mu\mathbf{n}$
G	$T,-p,\mathbf{n}$	$-S,-V,\mu$	$U-TS+pV$
Ω	T,V,μ	$-S,-p,-\mathbf{n}$	$U-TS+\mu\mathbf{n}$
Y	$S,-p,\mu$	$T,-V,-\mathbf{n}$	$U+pV+\mu\mathbf{n}$
0	$T,-p,\mu$	$-S,-V,-\mathbf{n}$	$U-TS+pV-\mu\mathbf{n}$

In a closed simple system with one component only it is possible to define eight different potentials, all of them having different sets of natural variables. As mentioned earlier, the natural variables come in conjugate pairs, with pairs of one intensive and one extensive variable. The eight potentials are listed in the table above. According to Alberty (1994), the intensive variables are in general easier to control in a laboratory experiment, and it is therefore convenient to have a thermodynamic potential with intensive variables as the natural variables. It is for instance easier to keep the temperature constant rather than the entropy. This is because there is standard laboratory equipment available for measuring the temperature, but no device for measuring entropy. The function U has only extensive variables as natural variables, namely $(S,V,\mathbf{n})$, while H and A have p and T respectively as the intensive variable. Last but not least, G has two intensive variables, T and p.

Equation 2.2 can be extended with more work terms, such as gravitational work ($\psi\,\mathrm{d}m$), work of electrical transport ($\sum_{i=1}^{n}\phi_i\,\mathrm{d}Q_i$), work of elongation ($f\,\mathrm{d}L$), surface work ($\gamma\mathrm{d}A_s$), work of electrical polarisation ($E\cdot\mathrm{d}p$), work of magnetic polarisation ($B\cdot\mathrm{d}m$), etc. The new terms add up in the same way as S, V and $\mathbf{n}$. This can be seen from the equation below.

$$\mathrm{d}U = T\,\mathrm{d}S - p\,\mathrm{d}V + \sum_{i=1}^{N}\mu_i\,\mathrm{d}N_i + \psi\,\mathrm{d}m + \sum_{i=1}^{n}\phi_i\,\mathrm{d}Q_i + f\,\mathrm{d}L + \gamma\mathrm{d}A_s + E\cdot\mathrm{d}p + B\cdot\mathrm{d}m \tag{2.8}$$

It is not required to use the complete Equation 2.8 in all engineering calculations, but rather use it as a starting point for identifying the most important terms of the current system. When more work terms are involved, a much larger number of Legendre transforms can be performed. Not all of them have found practical use, however.

Since the variables always appear in conjugate pairs, it is possible to construct $2^r - 1$ new thermodynamic potentials with the help of the Legendre transform. Here r is the number of independent variables, and -1 represents the starting potential. As mentioned earlier, it is possible to create eight different thermodynamic potentials for a simple one-component system. By starting with U, there are seven different Legendre transforms possible ($2^3 - 1 = 7$). Of these, only four have a universally accepted name. Enthalpy $H(S, p, \mathbf{n})$, Helmholtz energy $A(T, V, \mathbf{n})$ and Gibbs energy $G(T, p, \mathbf{n})$ are widely used in thermodynamics, while the grand canonical potential $\Omega(T, V, \mu)$ is used in statistical mechanics. The last three potentials have not found widespread use, but the potential $X(S, V, \mu)$ is exploited in Chapter 3 and 4, because it has some nice properties in multicomponent phase stability calculations. For a further discussion of all the mentioned work terms, see Alberty (1994).

2.1.1 Systematic Differentiation of Legendre Transforms

In this section, a systematic way of differentiating the Legendre transform to arbitrary order is derived. The starting point is the function $f^{(0)}(x_1, x_2, \ldots, x_n)$ from which the Legendre transform is used to change to alternative coordinates without loosing any information. The simplest transformation is: $f^{(1)}(\xi_1, x_2, \ldots, x_n)$. We shall next compare this function with the differential of $f^{(1)}(x_1, x_2, \ldots, x_n)$. The trick is to compare similar terms in order to systematize the differentiation.

$$\mathrm{d}f^{(1)}(\xi_1, x_2, \ldots, x_n) = \left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\cdots,x_n} \mathrm{d}\xi_1 + \sum_{i=2}^{n} \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{\xi_1, x_{j\neq i}} \mathrm{d}x_i \tag{2.9}$$

The total differential of $f^{(1)}$ is independent of the coordinate system and can also be written:

$$\mathrm{d}f^{(1)}(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f^{(1)}}{\partial x_1}\right)_{x_2,\cdots,x_n} \mathrm{d}x_1 + \sum_{i=2}^{n} \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} \mathrm{d}x_i \tag{2.10}$$

The two Equations 2.9 and 2.10 are compared to exclude common terms. First the total differential of the transformed variable is needed. By making use of Definition 2.4 we get:

$$\begin{aligned} \mathrm{d}\xi_1 &= \mathrm{d}\left(\frac{\partial f^{(0)}}{\partial x_1}\right)_{x_2,\ldots,x_n} \\ &= \left(\frac{\partial^2 f^{(0)}}{\partial x_1 \partial x_1}\right)_{x_2,\ldots,x_n} \mathrm{d}x_1 + \sum_{i=2}^{n} \left(\frac{\partial^2 f^{(0)}}{\partial x_i \partial x_1}\right)_{x_{j\neq 1,i}} \mathrm{d}x_i \end{aligned} \tag{2.11}$$

By substituting the result from 2.11 into 2.9, and comparing the result with Equation 2.10 gives:

$$\begin{aligned}
\mathrm{d}f^{(1)} &= \left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} \left(\frac{\partial^2 f^{(0)}}{\partial x_1 \partial x_1}\right)_{x_2,\ldots,x_n} \mathrm{d}x_1 \\
&+ \left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} \sum_{i=2}^{n} \left(\frac{\partial^2 f^{(0)}}{\partial x_i \partial x_1}\right)_{x_{j\neq 1,i}} \mathrm{d}x_i \\
&+ \sum_{i=2}^{n} \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{\xi_1, x_{j\neq i}} \mathrm{d}x_i \\
&= \left(\frac{\partial f^{(1)}}{\partial x_1}\right)_{x_2,\ldots,x_n} \mathrm{d}x_1 + \sum_{i=2}^{n} \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} \mathrm{d}x_i
\end{aligned} \tag{2.12}$$

Collecting similar terms in Equation 2.12, and the $\mathrm{d}x_1$ terms gives:

$$\begin{aligned}
\left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} \left(\frac{\partial^2 f^{(0)}}{\partial x_1 \partial x_1}\right)_{x_2,\ldots,x_n} &= \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} \\
\left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} &= \left(\frac{\partial^2 f^{(0)}}{\partial x_1 \partial x_1}\right)^{-1}_{x_2,\ldots,x_n} \left(\frac{\partial f^{(1)}}{\partial x_1}\right)_{x_2,\ldots,x_n}
\end{aligned} \tag{2.13}$$

For the $\mathrm{d}x_i$ terms it is possible to write a single sum. Since all the $\mathrm{d}x_i$ terms are independent, each term in the sum will have the same form. The sum only complicates the expressions, and is therefore omitted:

$$\left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} \left(\frac{\partial^2 f^{(0)}}{\partial x_i \partial x_1}\right)_{x_{j\neq 1,i}} + \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{\xi_1, x_{j\neq i}} = \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} \quad \forall\, i = 2..n \tag{2.14}$$

Equation 2.14 can now be solved using the result from Equation 2.13:

$$\begin{aligned}
\left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{\xi_1, x_{j\neq i}} &= \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} - \left(\frac{\partial^2 f^{(0)}}{\partial x_i \partial x_1}\right)_{x_{j\neq 1,i}} \left(\frac{\partial f^{(1)}}{\partial \xi_1}\right)_{x_2,\ldots,x_n} \\
&= \left(\frac{\partial f^{(1)}}{\partial x_i}\right)_{x_1, x_{j\neq i}} - \left(\frac{\partial^2 f^{(0)}}{\partial x_i \partial x_1}\right)_{x_{j\neq 1,i}} \left(\frac{\partial^2 f^{(0)}}{\partial x_1 \partial x_1}\right)^{-1}_{x_2,\ldots,x_n} \left(\frac{\partial f^{(1)}}{\partial x_1}\right)_{x_2,\ldots,x_n}
\end{aligned} \tag{2.15}$$

However, in order to write general expressions a more compact notation is needed. First, a function with two distinct groups of canonical variables is defined. The first group, $r_{i\in[1,m]}$, represents the group of variables that will be Legendre transformed, and the second group, $t_{i\in[m+1,n]}$, is not affected by the transform:

$$f(r_1, r_2, \ldots, r_m, t_{m+1}, t_{m+2}, \ldots, t_n) = f(r,t) \tag{2.16}$$

The Legendre transform of Equation 2.16 with respect to the variables in the first group can then be defined as:

$$g = f - f_r r = f - \sigma r \tag{2.17}$$

$$\sigma \equiv f_r \tag{2.18}$$

As for Equations 2.9 and 2.10 the same function shall be written in two different coordinate systems, with the objective of comparing the similar terms afterwards. The subscript r means that f is differentiated with respect to r:

$$h(\sigma_1, \sigma_2, \ldots, \sigma_m, \theta_{m+1}, \theta_{m+2}, \ldots, \theta_n) = h(\sigma, \theta)$$
$$h(r_1, r_2, \ldots, r_m, t_{m+1}, t_{m+2}, \ldots, t_n) = h(r, t)$$

The total differential of the function is independent of the coordinate system, and hence we can write:

$$\mathrm{d}h = h_\sigma \,\mathrm{d}\sigma + h_\theta \,\mathrm{d}\theta = h_r \,\mathrm{d}r + h_t \,\mathrm{d}t \tag{2.19}$$

The total differential of the transformed variable, being a partial derivative, is given from Equation 2.18 as:

$$\mathrm{d}\sigma = \mathrm{d}f_r = f_{rr} \,\mathrm{d}r + f_{tr} \,\mathrm{d}t \tag{2.20}$$

Note that θ is used only to distinguish the two expressions, and therefore

$$\mathrm{d}\theta \equiv \mathrm{d}t \tag{2.21}$$

When Equation 2.20 is inserted into 2.19, the total differential can be written:

$$\mathrm{d}h = h_\sigma (f_{rr} \,\mathrm{d}r + f_{tr} \,\mathrm{d}t) + h_\theta \,\mathrm{d}t = h_r \,\mathrm{d}r + h_t \,\mathrm{d}t \tag{2.22}$$

Similar terms in Equation 2.22 are compared to yield the following equations:

$$h_\sigma f_{rr} \,\mathrm{d}r = h_r \,\mathrm{d}r$$
$$h_\theta \,\mathrm{d}t + h_\sigma f_{tr} \,\mathrm{d}t = h_t \,\mathrm{d}t$$
$$h_r = h_\sigma f_{rr} \tag{2.23}$$
$$h_t = h_\theta + h_\sigma f_{tr} \tag{2.24}$$

Equations 2.23 and 2.24 constitute the transformation rules for a general Legendre transform, provided that the function is continuous and differentiable.

2.1.2 First Derivatives

The next task is to find the derivatives of the transformed equations in the original coordinates. The transformation rules from the previous section are used to achieve this. The transformed Equation 2.17 is differentiated to get:

$$g_r = f_r - f_{rr} r - f_r \delta = -f_{rr} r \tag{2.25}$$
$$g_t = f_t - f_{tr} r \tag{2.26}$$

When the conversion rules are applied, this yields:

$$g_\sigma f_{rr} = -f_{rr} r \tag{2.27}$$

$$g_\theta + g_\sigma f_{tr} = f_t - f_{tr} r \tag{2.28}$$

Expression 2.27 can be simplified, assuming that f_{rr} represents an invertible Hessian. If this assumption does not hold, the Legendre transform is undefined (typically a spinodal point of some kind). Thermodynamic potentials do in general yield invertible Hessians, but for some peculiar cases the Hessian is singular and therefore not invertible. Assuming that the Hessian f_{rr} is invertible, Equations 2.27 and 2.28 can be solved:

$$g_\sigma = -f_{rr}^{-1} f_{rr} r = -\delta r = -r \tag{2.29}$$

$$g_\theta = f_t - f_{tr} r - g_\sigma f_{tr} = f_t - f_{tr} r + r f_{tr} = f_t \tag{2.30}$$

2.1.3 Second Derivatives

Derivatives of g_σ:

Equation 2.29 differentiated with respect to r, gives $g_{r\sigma} = \delta$. With Rule 2.23 applied this yields $g_{\sigma\sigma} f_{rr} = -\delta$. The final result is written:

$$g_{\sigma\sigma} = -f_{rr}^{-1} \tag{2.31}$$

When the same equation is differentiated with respect to t, we get $g_{t\sigma} = 0$. By applying Rule 2.24 this yields $g_{\theta\sigma} + g_{\sigma\sigma} f_{tr} = 0$. If this result is combined with Equation 2.31 and reordered, the result is:

$$g_{\theta\sigma} = f_{rr}^{-1} f_{tr} \tag{2.32}$$

Derivatives of g_θ:

Equation 2.30 differentiated with respect to r, gives $g_{r\theta} = f_{rt}$. With Rule 2.23 applied this yields $g_{\sigma\theta} f_{rr} = f_{rt}$. The final result is written:

$$g_{\sigma\theta} = f_{rt} f_{rr}^{-1} \tag{2.33}$$

This is the same result as in Equation 2.32, which is also what one should expect since the order of differentiation is immaterial.

Finally, Equation 2.30 is differentiated with respect to t, which gives $g_{t\theta} = f_{tt}$. With Rule 2.24 this yields $g_{\theta\theta} + g_{\theta\sigma} f_{tr} = f_{tt}$. This result is next combined with Equation 2.32, and reordered to give:

$$g_{\theta\theta} = f_{tt} - f_{rr}^{-1} f_{tr} f_{tr} \tag{2.34}$$

Rules 2.23 and 2.24 can be applied repeatedly to obtain higher-order derivatives, and in Appendix A the third-order derivatives are defined. Of course the rules can be applied on these derivatives to obtain expressions of even higher order.

2.2 Stability

Thermodynamic stability is an important theoretical concept, which can be used to determine the stable phase equilibrium of a system. In the literature there are several different approaches to the stability test. Here, we shall make use of the approach of Callen (1985), which looks at the entropy function of a system. First the system is divided into two identical subsystems, and then energy is transfered from one subsystem to the other. If this transfer of energy does not give a raise in the total entropy for the system, it is deduced that the system is stable. An increase in entropy would lead to a phase transition. In Callen (1985) the definition of stability for the perturbation in U is given as:

$$S(U + \Delta U, V, \mathbf{n}) + S(U - \Delta U, V, \mathbf{n}) \leq 2S(U, V, n) \tag{2.35}$$

A graphical interpretation of this is shown in Figure 2.2. Criterion 2.35 is valid for all ΔU, but by using a three-point formula and letting $\Delta U \to 0$, the stability criterion reduces to:

$$\left(\frac{\partial^2 S}{\partial U^2}\right)_{V,\mathbf{n}} \leq 0 \tag{2.36}$$

Criterion 2.36 is less restrictive than Criterion 2.35, since the latter holds only for small ΔU. If Criterion 2.36 is fulfilled but not Criterion 2.35, the system is said to be in a metastable state, and that 2.36 corresponds to a local stability criterion.

From a geometrical point of view, we can say that the Criterion 2.35 is valid in the concave region of the curve. The reason for the concavity of the entropy curve, comes from the extremum of the curve, which tells us that the entropy tends towards a maximum. The necessary and sufficient condition of a maximum state is that the first derivative is zero (an extremum), and that the second derivative is negative (maximum). Another way of interpreting the stability criterion is to look at the tangent. The system is stable if the tangent to a point on the graph does not intersect the curve (Luenberger, 1984).

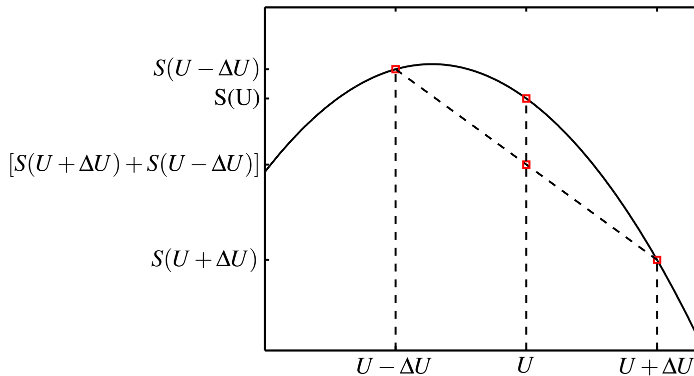


Figure 2.2: The average entropy of the two subsystem is lower than the original system. The system is therefore stable.

In Figure 2.3, the solid tangent lines are everywhere above the curve, and they thereby represents the equilibrium state of the system (convex hull). The tangent line intersecting at points A and D represents the stability limit of the curve. In the region ABCD criterion 2.35 is violated, and this region is therefore globally unstable. The regions AB and CD (dashed region in Figure 2.3) are only locally stable (metastable), while the region BC is everywhere unstable (dotted region in Figure 2.3). In region ABCD the system can* split into two phases, where one phase will attain state A, and the other will attain state D. The sizes of each of the phases are given relative to the AED line. The dashed tangent lines in Figure 2.3 represents the tangent lines in the metastable region. As one can see these lines intersect the curve, which means that the point is not stable.

The entropy curve was used here only as an example to show the basic concepts. It is straightforward to write the same criterion for internal energy U. The only difference is the fact that the extremum of the internal energy is a minimum instead of a maximum, and therefore the inequality changes. Criterion 2.35 can also easily be extended to other state variables. Geometrically, the stability criteria require that the hyper-surface of the thermodynamic potential lies everywhere above or below† the tangent hyper-planes.

Modell and Reid (1983) use a different approach. They consider a simple isolated system with one phase. The system is in equilibrium, hence $\delta S = 0$. Next they divide the isolated system into two parts, one large part, α, and one small part, β. This is used to derive the stability criterion. A general stability criterion can be

*In the unstable region BC the phase split will occur

†depending on the inequality

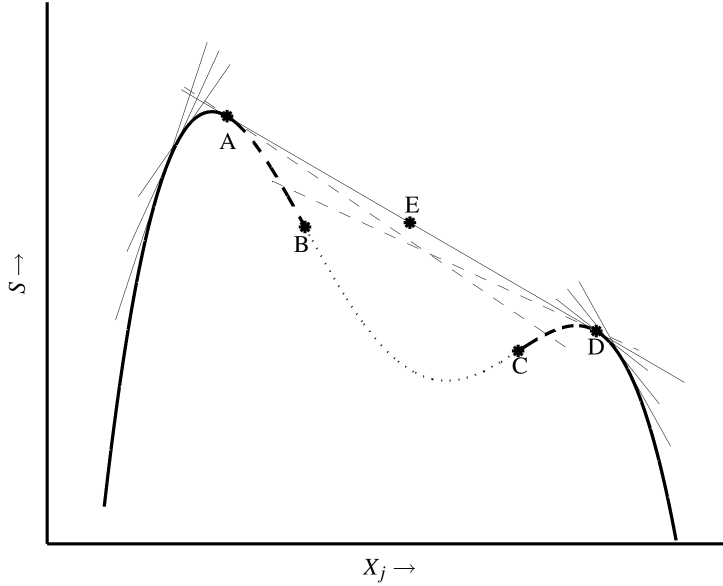


Figure 2.3: Stability regions of a function, where a solid line corresponds to a stable region, a dashed line corresponds to a metastable region and a dotted line corresponds to a unstable region.

written as:

$$\delta^2 y^{(0)} = K \sum_{i=1}^{m} \sum_{j=1}^{m} y_{ij}^{(0)} \delta x_i \delta x_j > 0 \tag{2.37}$$

Where K is a positive constant and m is the number of components (n) + 2. For a stable system $\delta^2 y^{(0)}$ is positive, and for an unstable system $\delta^2 y^{(0)}$ is negative. This means that where the curvature of the energy function is convex, the system is stable, and where the curvature is concave, the system is unstable. This is equivalent to Criterion 2.35 mentioned in the previous paragraph. When the second derivative is zero it is necessary to study the higher-order derivatives in order to determine the stability of the system. If the third derivative is positive the system is stable, and if it is negative the system is unstable. Criterion 2.37 implies that y^0 must be positive for all possible variations of δx. For example, for $U(S,V,N_1 \ldots N_j)$ all perturbations in $\delta S, \delta V, \delta N_j$ must give a positive change in $\delta^2 U$. This leads to the criteria shown in Equation 2.38, which are equivalent to the criteria given by both Bartis (1973) and Beegle et al. (1974b):

$$y_{kk}^{(k-1)} > 0 \quad k = 1, \ldots, m-1 \tag{2.38}$$

Table 2.2: Different possible stability criteria for a one component system.

Variable order	Stability criterion
$(S,V,\mathbf{n})$	$A_{VV} = -(\partial P/\partial V)_{T,\mathbf{n}} > 0$
$(S,\mathbf{n},V)$	$A_{\mathbf{nn}} = (\partial \mu/\partial \mathbf{n})_{T,V} > 0$
$(V,S,\mathbf{n})$	$H_{SS} = (\partial T/\partial S)_{P,\mathbf{n}} > 0$
$(V,\mathbf{n},S)$	$H_{\mathbf{nn}} = (\partial \mu/\partial \mathbf{n})_{P,S} > 0$
$(\mathbf{n},S,V)$	$U`_{SS} = (\partial T/\partial S)_{\mu,V} > 0$
$(\mathbf{n},V,S)$	$U`_{VV} = -(\partial P/\partial V)_{\mu,S} > 0$

Here, $y^{(k-1)}$ means the k-1 Legendre transform of $y^{(0)}$, where $y^{(0)}$ in our context is $U(S,V,\mathbf{n})$. The subscript means that y is differentiated twice with respect to the k'th variable. The last Legendre transform in Equation 2.38 is by definition zero* and is therefore not of interest here. At the stability limit all the criteria in Equation 2.38 are zero, but it can be proved that the criterion given in Equation 2.39 is violated first.

$$y^{(m-2)}_{(m-1)(m-1)} > 0 \tag{2.39}$$

This is therefore a sufficient criterion for local stability.

The order of the variables in $y^{(0)}$ is not fixed, which means that Equation 2.39 will change with the ordering of the variables. This gives several different criteria, which are all equivalent. For curiosity all possible criteria for a one component system is shown in Table 2.2 (Beegle et al., 1974b). The stability criteria mentioned here are important for the discussion of critical points in the next section.

2.2.1 Critical points

Critical points are closely related to the stability discussed in Section 2.2, as the critical point lies on the stability limit. In fact, a critical point of a two-phase boundary is where the binodal curve is tangent to the spinodal curve. Hence, at the critical point the properties of two phases are indistinguishable, which means the spinodal curve goes through an extremum tangent to the binodal curve in the critical point. At the critical point, the stability criteria in Equation 2.39 reduce to zero. An additional criterion is therefore required to determine the stability. The stability requirement remains the same at the critical point, and we therefore have to look at higher-order derivatives to determine whether the critical point is

*An Euler homogeneous function of first order is always zero when it is transformed with respect to all its variables. This is a property of Euler homogeneous functions and not the Legendre transform.

a stable one. Reid and Beegle (1977) gives the general criteria for an n'th order critical point:

$$\begin{aligned} y^{(m-2)}_{(m-1)} &= 0 \\ y^{(m-2)}_{(m-1)(m-1)} &= 0 \\ y^{(m-2)}_{(m-1)(m-1)(m-1)} &> 0 \end{aligned} \tag{2.40}$$

For higher-order critical points, the criteria in Equation 2.40 is increased by two more derivatives for each order. For instance, at the tri-critical point the first four derivatives are zero, and the fifth derivative is positive.

Chapter 3

Phase Stability Test

In this chapter a new method for checking a phase assembly to verify that it is globally stable is presented. This is done by introducing a trial phase to check whether this can lower the energy of the phase assembly. If the introduction of this trial phase lowers the total energy of the system, it means that there is another phase assembly which is more stable. In the literature several different approaches for phase stability tests have been presented. Most of them utilise the tangent plane distance proposed by Gibbs (1876) and proved by Baker et al. (1982). Michelsen (1982a) proposed a method for introducing trial phase in order to reduce the total Gibbs energy of the phase assembly. This is essentially a local method, but has proved to be very effective. Parts of the work done by Michelsen (1982a) has been reproduced here by using a similar tangent plane distance method. Alternatives to the tangent plane distance method are for instance global optimization (McDonald and Floudas, 1995) and parameter continuation (Sun and Seider, 1995). These approaches were not considered in this work.

The method presented here attempts to verify the phase assembly at the same intensive conditions as the initial state by minimizing the energy potential $X(S,V,\mu)$ at constant $\mu_1,\dots,\mu_n$. The advantage of this potential compared to Gibbs energy is that it contains only two extensive variables. It is thereby possible to explore the thermodynamic space spanned by the potential X by iterating two extensive variables.

This is possible because the chemical potential of each component must be the same in all stable phases in equilibrium with the initial phase, see the funneling theorem by Branin (1972). The practical calculations are done by defining a tangent plane distance of zero at the starting point, and then calculate the tangent

plane distance for all other point. All points which have a tangent plane distance less than zero indicate that a more stable phase assembly exists.

The method is not using standard optimization tools, instead it traverses the thermodynamic space spanned by the variables S and V using a stepping procedure. There are two major challenges with this approach: First, the thermodynamic model is the Soave–Redlich–Kwong (SRK) equation of state (Soave, 1972). This is a Helmholtz energy model, which is explicit in the variables T, V, N_i rather than S, V, μ_i. This means that the steps taken in the variables S, V must be mapped into to the original model coordinates. The other major challenge is to control the step size. The method should move along the trajectory as fast as possible, but not so fast that any of the local minima are missed, or is unable to converge to the constant μ-manifold. A set of heuristic rules is implemented in order to facilitate this.

Several different approaches are used in this chapter, but no direct comparison of the computation speeds for the alternatives will be presented. The reason for this is that the different methods have been developed at different times in the project, and with different implementations of the thermodynamic model. A comparison in speed is therefore not fair. The focus in this chapter is therefore not speed, but rather problem solving in a satisfactory and reliably way.

3.1 Example from Literature

When a new method is under development, it is wise to make sure that the starting point is a sound one. It was therefore decided to reproduce Figure 5 of Michelsen (1982a), to make sure the models are comparable. Whilst the paper states that the SRK equation of state is used, it does not say anything about the parameters. For the system in question (CH_4–H_2S) several different parameter sets were tried, and the one which gave the composition–energy curve that looked most like the one reported, is used. No further efforts were made to make an exact match. The parameters used here, are given in Appendix B.

Figure 3.1 shows the molar Gibbs energy versus composition at a constant temperature of 140K and a pressure of 40 atm. As already mentioned, the SRK-model is explicit in T, V, $\mathbf{n}$, but not in p, and thus it is necessary to solve the equations iteratively for the pressure in every point on the curve. Chapter 6–8 describes the tool with which the SRK equations were implemented, giving easy access to both the gradient and the Hessian of the Helmholtz model. The correct pressure is therefore solved by using Newton's method, where the pressure in each iteration is given by

$\left(\frac{\partial A}{\partial V}\right)_{T,\mathbf{n}} = -p$:

$$p = p_\circ + (V_{n+1} - V_n)\left(\frac{\partial p}{\partial V}\right) + O((\Delta V)^2) \tag{3.1}$$

The above equation is solved with respect to $\Delta V = (V_{n+1} - V_n)$, to yield an update in V becomes:

$$\Delta V = (p - p_\circ)\left(\frac{\partial^2 A}{\partial V^2}\right)^{-1}. \tag{3.2}$$

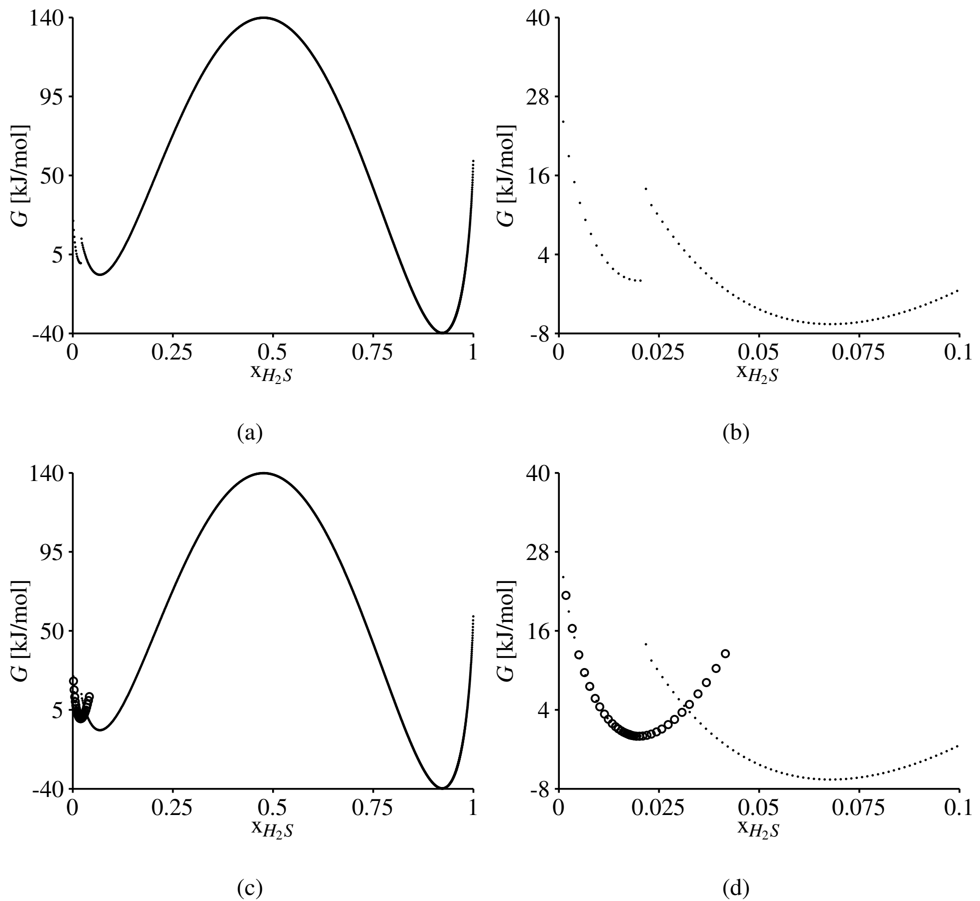


Figure 3.1: Mole fraction of H_2S plotted against molar Gibbs energy for the system CH_4–H_2S. The computations are started in both the leftmost (circles) and rightmost (dots) minima.

The Gibbs energy is calculated by the relation $G = \sum N_i \mu_i$, where $\mu_i = \left(\frac{\partial A}{\partial N_i}\right)_{T,V}$. The easiest way to span the curve is to continuously change the composition from

one pure component to the other. This will give the seemingly discontinuous curve shown in Figure 3.1, which is also shown by Michelsen (1982a). The reason for this behaviour is that there are multiple solutions in parts of the composition space. This can be seen from Figure 3.1(d) where the two branches of $G(x)$ intersects at $x \simeq 0.03$. The entire curve is obtained by starting on each side of the discontinuity. However, it is a bit unsatisfactory to not be unable to explore the whole curve in one stroke, and a more sophisticated method was therefore developed. The null space of the matrix shown below gives a step direction, which will trace out the whole curve, as shown in Figure 3.2.

$$\begin{pmatrix} 1 & 0 & 0 \\ A_{TV} & A_{VV} & A_{\mathbf{n}V} \\ 0 & V & \mathbf{n} \end{pmatrix} \begin{pmatrix} \delta T \\ \delta V \\ \delta \mathbf{n} \end{pmatrix} = 0 \tag{3.3}$$

The first row of the matrix in Equation 3.3 explicitly keeps the temperature constant. The second row impose a change in volume and mole numbers such that the effect on the pressure is minimal. The last row makes sure the changes does not scale the system. The direction of the null space is controlled such that the

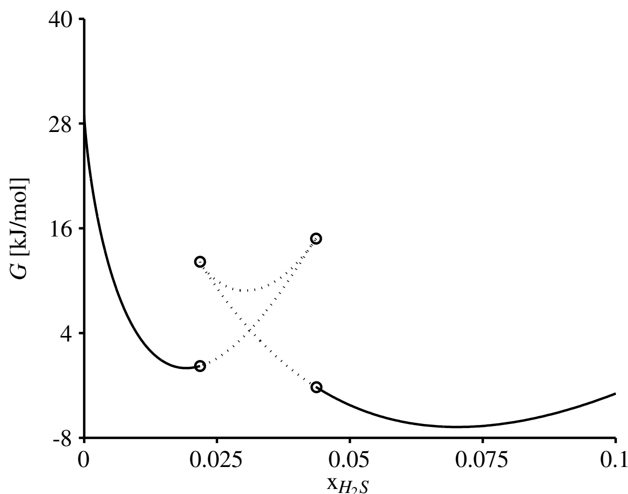


Figure 3.2: Swallowtail in Gibbs energy for the system CH_4–H_2S, at T = 140K and p = 40 atm. The circles and the dotted line show the region where the swallowtail occurs.

principal direction remains the same from iteration k to $k+1$ along the curve. The swallowtail in Figure 3.2 is explained by the catastrophe theory of Thom (1975) and Zeeman (1977), and is due to the folding of the multidimensional space of which this figure is a projection. When pursuing the null space solution in one

direction, the solution seemingly switches from one solution branch to another at two cusp points giving rise to the hysteresis lines shown. Later in this chapter it is shown that the changes in the extensive variables are continuous and that the discontinuity is therefore in the projection only.

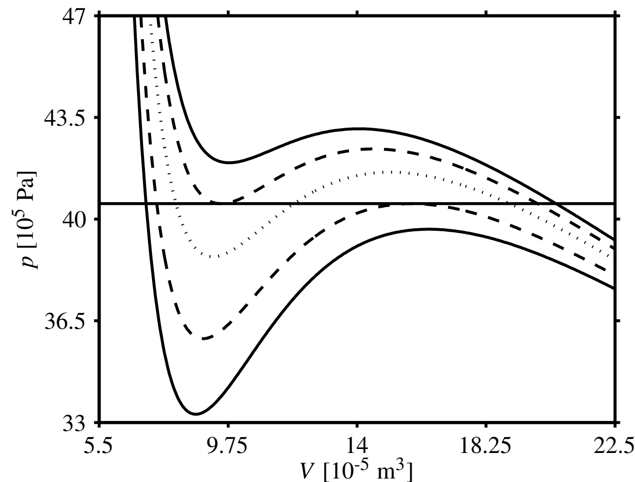


Figure 3.3: p–V curves at different temperatures for the system CH_4–H_2S. The horizontal line corresponds to 40 atm.

The multiple solutions can be understood by plotting p–V curves for different temperatures. The straight line shown in Figure 3.3 represents the constant pressure of 40 atm, which is the pressure along the manifold projection in Figure 3.2. The solid lines of both Figure 3.2 and Figure 3.3 correspond to a region with only one solution. The circles in Figure 3.2, and dashed lines in Figure 3.3 corresponds to points with two solutions. The dotted lines correspond to a region with three different solutions for each mole fraction. When moving through the region of several solutions, the behaviour of the p–V curves can be explained in the following manner: The curve will move from a region where the p–V curve intersects the 40 atm line at one point, to a tangent line where there are two identical volumes for the given pressure. It will further be three solutions to the volume, until the constant pressure curve again becomes a tangent to the p–V curve, thereby giving two identical solutions. This phenomenon will be investigated further in the next section.

3.2 Alternative Energy Function Formulation

The tangent plane distance method shown in the previous section works fine for a two-component system, but it cannot be extended to systems with more than two components. The reason for this is that the complexity of the projections made by changing the compositions systematically gets bigger fast, and besides, the plot will be multi-dimensional. This also means that it is not possible, or at least not desirable, to span the entire thermodynamic space of Gibbs energy in order to find the stable phase assembly.

Even though not every possible alternative is tested, local methods, like the one of Michelsen (1982a), and different forms of global optimization methods seem to work satisfactory in most cases. These methods do not check all possible states in the search for the stable phase, which is obviously much faster than traversing the whole thermodynamic space. The only advantage of the approach presented here is that, given small step sizes, the most stable minimum will always be found. However, this requires a thermodynamic potential which makes it possible to easily span the entire thermodynamic space. This task is solved by using a thermodynamic potential with only two extensive variables. The only real alternative is the unnamed energy potential $X(S,V,\mu)$, since it is undesirable to use an energy potential where the chemical components are split into two disjoint sets.

The thermodynamic potential X is defined by using the Legendre transforms as shown in Chapter 2. Compared to the Gibbs energy potential of the previous section, all the variables need to be transformed. For Gibbs energy the space is spanned by using mole fractions. Mole fractions cannot be used for X, but a dimensionless variable $\alpha \in [0,1]$ is created instead. This variable is defined as

$$\alpha = \frac{|T_\circ|S}{|T_\circ|S+|p_\circ|V}, \tag{3.4}$$

where $\lim_{S\to 0} \alpha = 0$ and $\lim_{V\to 0} \alpha = 1$. Changing α corresponds to moving along a straight line from zero entropy to zero volume in the S–V plane. This also means that the denominator of α is constant, and needs to be calculated only once at the starting point. As explained before V is an explicit variable in the equation of state being used. Since the constant intensive properties are defined by the phase assembly that are going to be tested, it is natural to start there. This will never be at the endpoints of α, which means the α space must be explored in both directions from the starting point.

The thermodynamic model used here is exactly the same as for the Gibbs energy case. This model is a Helmholtz energy model with the free variables T, V and $\mathbf{n}$. This means that V is the only explicit variable in the problem since this is the

only variable that can be used directly in the thermodynamic model. All the other variables must be mapped back to the Helmholtz variables by solving a system of Jacobian equations with Newton's method. Since the denominator of α is constant, it is possible to make a step in α, and calculate the corresponding S and V. The tangent plane distance defined for this energy potential, will be at constant chemical potentials. This means that the Helmholtz model must be updated such that the entropy is in accordance with α, and that the chemical potentials are kept constant. The problem to be solved gives rise to the following equations:

$$\begin{pmatrix} S_T & S_{\mathbf{n}} \\ \mu_T & \mu_{\mathbf{n}} \end{pmatrix} \begin{pmatrix} \delta T \\ \delta \mathbf{n} \end{pmatrix} = \begin{pmatrix} S - S(\alpha) \\ \mu - \mu^\circ \end{pmatrix} \tag{3.5}$$

Since entropy and chemical potentials are defined by the gradient of Helmholtz energy, the Hessian of Helmholtz energy can be used in the Jacobian of the previous equation to calculate the update in model variables:

$$\begin{pmatrix} \delta T \\ \delta \mathbf{n} \end{pmatrix} = \begin{pmatrix} A_{TT} & A_{T\mathbf{n}} \\ A_{T\mathbf{n}} & A_{\mathbf{nn}} \end{pmatrix}^{-1} \begin{pmatrix} S - S(\alpha) \\ \mu - \mu^\circ \end{pmatrix} \tag{3.6}$$

This will constrain the system to the constant chemical potential manifold, while the Gibbs energy problem of the previous section constrained the system to the constant temperature and pressure manifold. This creates a basis for finding the point on the constant chemical potential manifold which has the lowest energy. The logical use of this approach is to test whether the trial phase is stable or not. To do this, a tangent plane distance method is used. The unbounded tangent plane is given by:

$$\tilde{X} = (T - T_\circ)S - (p - p_\circ)V \tag{3.7}$$

To make it dimensionless the following merit function is given:

$$\tilde{x} = \frac{\tilde{X}}{|T_\circ|S + |p_\circ|V} \equiv \frac{T - T_\circ}{|T_\circ|}\alpha + \frac{p - p_\circ}{|p_\circ|}(1 - \alpha) \tag{3.8}$$

Note that the tangent plane distance ($\tilde{x}$) by definition is zero at the starting point. If $\tilde{x} \geq 0, \forall \alpha \in \langle 0, 1 \rangle$, the original system is stable. If $\tilde{x} < 0, \exists \alpha \in \langle 0, 1 \rangle$ the original system is not globally stable, see Section 2.2 for details on stability.

As mentioned earlier, the main objective here is to verify that the starting point is globally stable on the constant chemical potential manifold. In Equations 3.4–3.8 a way of doing this in just one variable, instead of multiple variables is established. What remains, is to find an appropriate way to vary α such that no possible minimum is missed, and without spending too much computer resources. The first approach was to keep a constant step in α. As can be seen from Figure 3.4 this simple rule makes it possible to trace out the curve nicely. Since the starting point

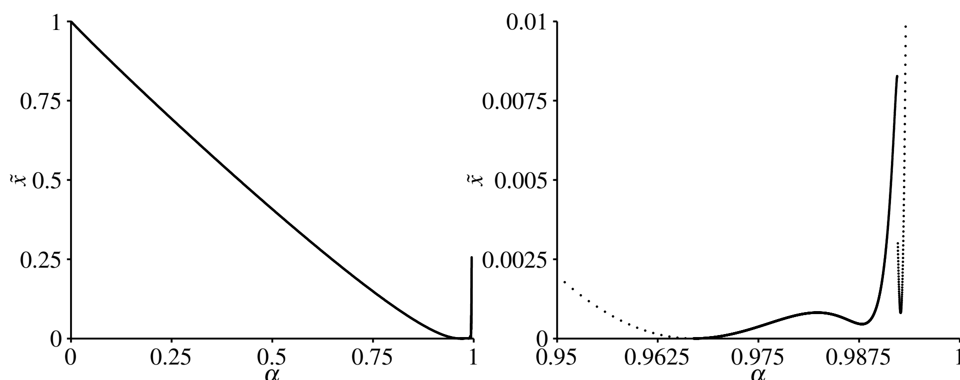


Figure 3.4: Projection of tangent plane distance $\tilde{x}$, as a function of α for the system CH_4–H_2S. The left figure shows the whole curve, while the right figure only shows the area where the minima occur.

will always be somewhere along the line, the method must step in both directions from the starting point to trace out the whole curve. The problem with this approach, however, is that it pays no attention to the real task, which is to find the minima with an energy which is lower than at the starting point, i.e. that have negative tangent plane distances. To make sure no minimum is missed the method must take rather short steps which makes it rather slow.

3.2.1 Controlling the step size

A better approach than using small steps is to take relatively large steps, and then use some extra effort in locating the minima exactly. This is achieved by a Taylor expansion in $\tilde{x}$ with respect to α to the second order:

$$y = \tilde{x} + h\tilde{x}_\alpha + \frac{h^2}{2}\tilde{x}_{\alpha,\alpha} \tag{3.9}$$

The differentiation of $\tilde{x}$ is not straightforward, and is given below. To make the derivation simpler $\tilde{X}$ is used instead of $\tilde{x}$, and the result is scaled afterwards. The total differential of $\tilde{X}$ is given by:

$$\mathrm{d}\tilde{X} = \mathrm{d}TS + (T - T_\circ)\,\mathrm{d}S - \mathrm{d}pV - (p - p_\circ)\,\mathrm{d}V \tag{3.10}$$
$$\mathrm{d}\tilde{X} = (T - T_\circ)\,\mathrm{d}S - (p - p_\circ)\,\mathrm{d}V \tag{3.11}$$

The simplification is due to the Gibbs-Duhem equation, which states that $S\,\mathrm{d}T - V\,\mathrm{d}p + \sum_i N_i\,\mathrm{d}\mu_i = 0$. Since μ is constant along the manifold the last term in the

Gibbs-Duhem equation equals zero. S and V are plain functions of α:

$$S = \frac{\alpha C}{|T_\circ|} \tag{3.12}$$

$$V = \frac{(1-\alpha)C}{|p_\circ|}, \tag{3.13}$$

and their total differentials are easily defined:

$$\mathrm{d}S = \frac{\partial S}{\partial \alpha}\,\mathrm{d}\alpha = \frac{C}{|T_\circ|}\,\mathrm{d}\alpha \tag{3.14}$$

$$\mathrm{d}V = \frac{\partial V}{\partial \alpha}\,\mathrm{d}\alpha = -\frac{C}{|p_\circ|}\,\mathrm{d}\alpha \tag{3.15}$$

A combination of Equations 3.14, 3.15 and 3.11 gives:

$$\frac{\mathrm{d}\tilde{X}}{\mathrm{d}\alpha} = \frac{(T-T_\circ)C}{|T_\circ|} + \frac{(p-p_\circ)C}{|p_\circ|} = C\left(\frac{T}{|T_\circ|} + \frac{p}{|p_\circ|} - 2\right) \tag{3.16}$$

The second-order total differential is given by:

$$\mathrm{d}^2\tilde{X} = \mathrm{d}T\,\mathrm{d}S - \mathrm{d}p\,\mathrm{d}V \tag{3.17}$$

T and p are conceivable functions of S and V, and the total differentials may be written as (implicit functions):

$$\mathrm{d}T = \frac{\partial T}{\partial S}\,\mathrm{d}S + \frac{\partial T}{\partial V}\,\mathrm{d}V \tag{3.18}$$

$$\mathrm{d}p = \frac{\partial p}{\partial S}\,\mathrm{d}S + \frac{\partial p}{\partial V}\,\mathrm{d}V \tag{3.19}$$

The second-order differential can then be written as:

$$\mathrm{d}^2\tilde{X} = \frac{\partial T}{\partial S}(\mathrm{d}S)^2 + \frac{\partial T}{\partial V}\,\mathrm{d}V\,\mathrm{d}S - \frac{\partial p}{\partial S}\,\mathrm{d}S\,\mathrm{d}V - \frac{\partial p}{\partial V}(\mathrm{d}V)^2 \tag{3.20}$$

By inserting the total differential of S and V into Equation 3.20, the second-order derivative is given by:

$$\frac{\mathrm{d}^2\tilde{X}}{\mathrm{d}\alpha^2} = \frac{\partial T}{\partial S}\left(\frac{C}{|T_\circ|}\right)^2 - \frac{\partial T}{\partial V}\frac{C^2}{|p_\circ||T_\circ|} + \frac{\partial p}{\partial S}\frac{C^2}{|T_\circ||p_\circ|} - \frac{\partial p}{\partial V}\left(-\frac{C}{|p_\circ|}\right)^2 \tag{3.21}$$

Figure 3.5 show how the second-order Taylor expansion develops along the curve. Both $\frac{\mathrm{d}\tilde{X}}{\mathrm{d}\alpha}$ and $\frac{\mathrm{d}^2\tilde{X}}{\mathrm{d}\alpha^2}$ are used in addition to the function value of the Taylor expansion to detect a minimum. If $\frac{\mathrm{d}\tilde{X}}{\mathrm{d}\alpha}$ in the direction the step is negative, and then turns positive in the next iteration, a minimum will be located between these two points. The Taylor expansion in Equation 3.9 is differentiated with respect to α, and if the sign of the differential changes, it means that the Taylor expansion goes through an extremum. By watching $\frac{\mathrm{d}^2\tilde{X}}{\mathrm{d}\alpha^2}$, it can be decided whether it is a minimum or a maximum. If the Taylor expansion indicates a minimum in the forward direction,

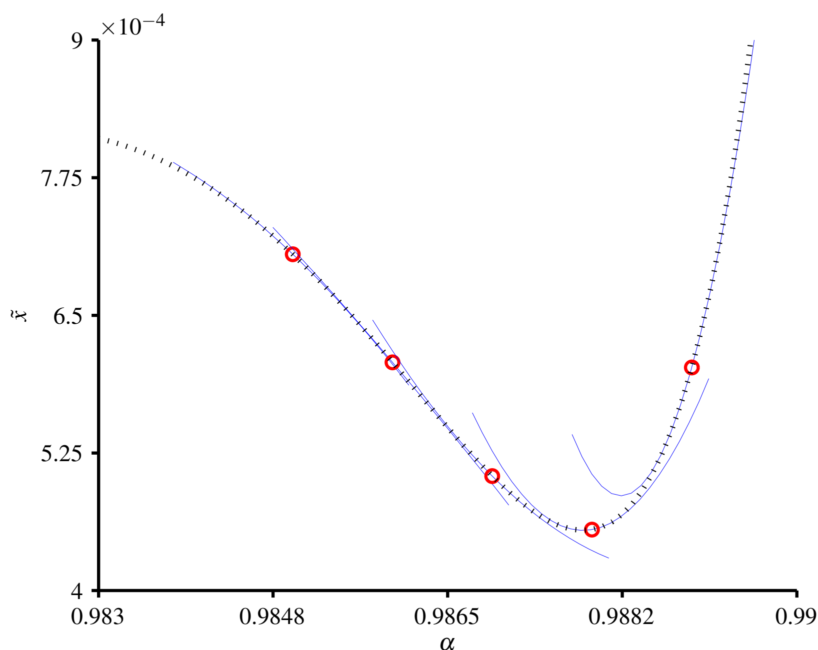
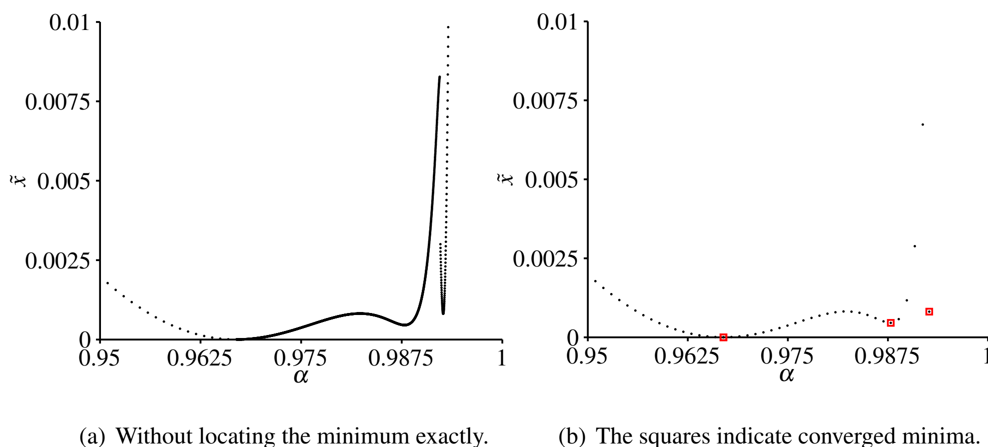


Figure 3.5: Taylor expansion (lines) shown in each point (circles) of projection of tangent plane distance $\tilde{x}$ as a function of α for the system CH_4–H_2S. The dotted line shows the curve as shown in Figure 3.4.

the step size is decreased accordingly, but no effort is made to locate the minimum exactly. It could be tempting to go straight for the minimum, but experience shows that it is better to just decrease the step size, once a minimum is predicted. If, on the other hand, the Taylor expansion shows that the method has passed a minimum, it will try to locate this. The Taylor expansion is spanned exactly one step in each direction. This will prevent the method from detecting a converged minimum, which is just passed, and at the same time predict a possible minimum within the next step. If no minimum is predicted, or detected, the step size is increased.

Locating the exact minimum is done in a inner-outer loop Newton iteration. The first and second derivatives of $\tilde{x}$ with respect to α are used to converge the minimum. In the inner loop the constant μ manifold is solved. When the change in α goes to zero, the minimum tangent plane distance has converged. By this approach it is possible to take larger steps without overlooking a minimum, as can be seen from Figure 3.6. It should be noted that the minimum can also be solved simultaneously in a single Newton loop. This is not attempted here since no stability problems were experienced with the double loop.



(a) Without locating the minimum exactly. (b) The squares indicate converged minima.

Figure 3.6: Comparison of projection of tangent plane distance $\tilde{x}$ as a function of α for the system CH_4–H_2S with and without converged minima.

Finding the optimal step size for an iteration problem is not easy. In one way or the other heuristic rules must be used. These rules will obviously work better for some cases than others. Since the aim here is to construct a general* method, the user is given a small flexibility concerning the step size. The maximum, the minimum and the initial step sizes are required as input to the method. The internal step size controller is not allowed to go outside these boundaries. Each time a minimum is converged the step size is set back to the initial step size. If the step size becomes smaller than the minimum step size without converging to the manifold, the method will stop. The step size parameters enables the user to set the necessary resolution for the problem. Finding the optimal step size for a given problem has to be done by trial and error. Large steps are not necessarily faster than smaller steps, since the internal step size control may reduce the step size repeatedly in order to be able to converge to the manifold, and a smaller step might actually be faster.

3.2.2 Catastrophe handling

There is no way to avoid the catastrophe that occurred in the Gibbs energy projection shown in Figure 3.2. Since the projection of the swallowtail is not monotonically increasing in α, the method described in Section 3.2 will not work in the general case. The method can step across the area where the catastrophe occurs,

*It is here meant general as in the same type of problems as presented here, not general purpose

and thereafter converge to a solution on the other side. The method may therefore fail, resulting in a dead lock. An alternative approach is therefore needed, and the swallowtail shown in Figure 3.2 is traced out by pursuing a null space of Equation 3.3. A similar approach with updates in both T,V,$\mathbf{n}$ and S,V,$\mathbf{n}$ were also examined for the problem in $X(S,V,\mu)$. Both approaches experience stability problems when the step size is increased, and to gain better insight into the problem further investigations were required.

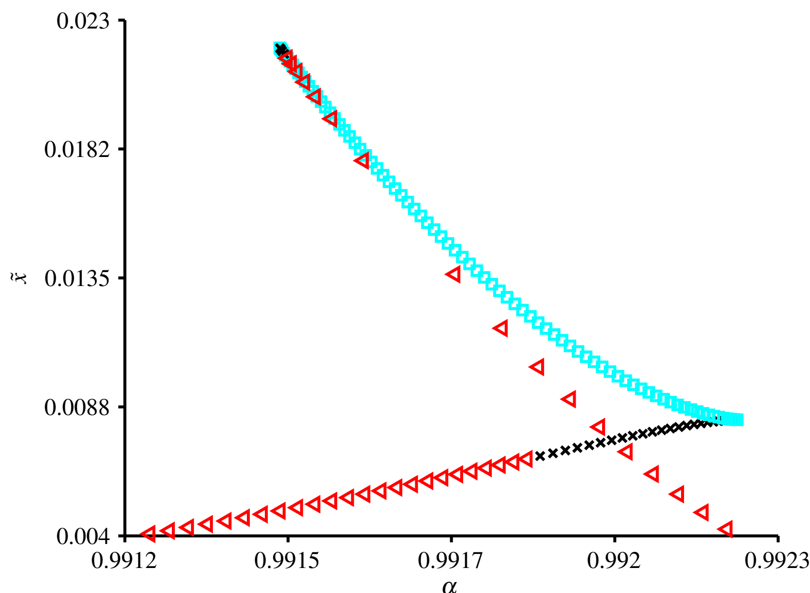


Figure 3.7: Swallowtail in $X(S,V,\mu)$ for the system CH_4–H_2S. The squares, triangles and crosses corresponds respectively to one, two or three negative eigenvalues in X_{VV} and $X_{\mu\mu}$.

Even though the projection in $\tilde{x}$,α experiences abrupt changes, the same is not true for other projections as shown in Figures 3.8 and 3.9. The circles in these figures indicate the cusp points shown in Figure 3.7. An aid for solving the problem is to watch the number of negative eigenvalues as the system state changes. Figure 3.7 shows that the number of negative eigenvalues changes at the cusp points, and this can explain why the simple method experience problems in these points. A null space direction gives only a linearization of the next step since the chemical potentials, which are constant on the manifold, are nonlinear functions of the state variables and must therefore be converged in each step.

To be able to handle the curvature better, a higher-order prediction can be used

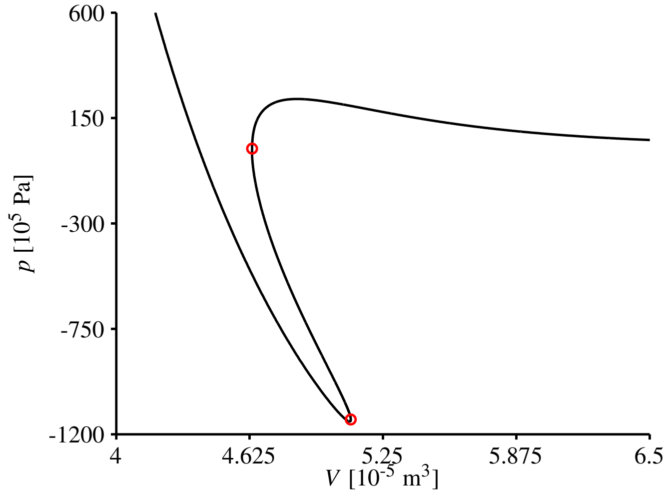


Figure 3.8: Projection of p–V at constant S and μ for the system CH_4–H_2S. The circles indicate the cusp points from Figure 3.7.

based on a Taylor expansion in T and $\mathbf{n}$. V is by definition correct since it is explicitly predicted and updated. The task is to move along a constant μ-manifold by changing S and V, and the higher-order Taylor expansion is used to find a good prediction in the T and $\mathbf{n}$ variables. In the previous subsection a monotonically increasing α was used to calculate a step in S and V, and the constant μ-manifold was converged according to the updated values of S and V. In Figure 3.7 a null space is used to find steps in S and V, defined as:

$$\begin{pmatrix} S & V & \mathbf{n} \\ \frac{\partial \mu}{\partial S} & \frac{\partial \mu}{\partial V} & \frac{\partial \mu}{\partial \mathbf{n}} \end{pmatrix} \begin{pmatrix} \Delta S \\ \Delta V \\ \Delta \mathbf{n} \end{pmatrix} = 0 \tag{3.22}$$

This indicates a direction which is orthogonal to the current state S, V and $\mathbf{n}$, that is the maximum change in the extensive state while μ is constant. The model needs updates in T, V and $\mathbf{n}$, which means the prediction must be mapped back to these coordinates. The problem is that this must be done at constant μ, and since this is a nonlinear mapping it is not possible to do it exact. It was therefore chosen to use a third-order Taylor expansion for this purpose. V is explicit in both formulations and the prediction of V is by definition correct. The combined prediction of S and V give a step in α and is therefore kept, while $\mathbf{n}$ is recalculated based on the Taylor expansion. The updates in S and V calculated from the null space in Equation 3.22

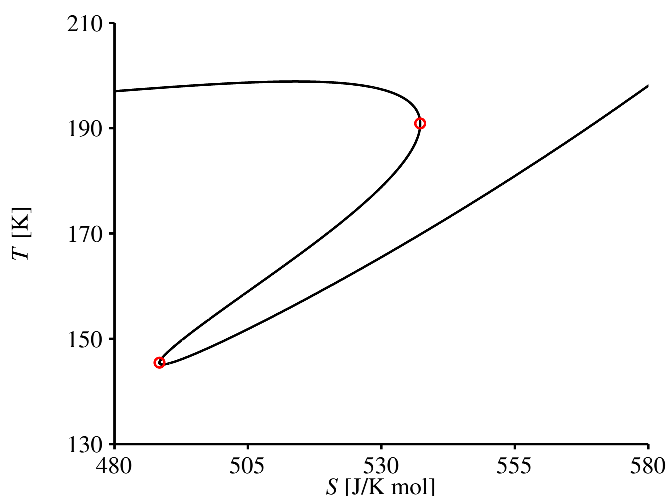


Figure 3.9: Projection of T–S at constant V and μ for the system CH_4–H_2S. The circles indicate the cusp points from Figure 3.7.

are:

$$\Delta S = V - \mathbf{n}^{\mathrm{T}} \left(\frac{\partial \mathbf{n}}{\partial -S} \right)_{\mu,V} \tag{3.23}$$

$$\Delta V = -S - \mathbf{n}^{\mathrm{T}} \left(\frac{\partial \mathbf{n}}{\partial V} \right)_{\mu,S} \tag{3.24}$$

However, since Matlab gives an orthonormal basis for the null space, the updates in S and V must be scaled for further use, but at the same time the prediction in $\mathbf{n}$ must also be scaled. The mole numbers are therefore calculated as a first-order Taylor expansion in S and V according to the following equation, where $X_S \hat{=} T$ and $X_\mu \hat{=} n$:

$$\Delta \mathbf{n} = -X_{S\mu} \Delta S - X_{V\mu} \Delta V \tag{3.25}$$

The updates in S, V and $\mathbf{n}$ can now be scaled such that an appropriate step is taken, where the maximum relative allowed step change is determined by the user. The scaled steps in S and V are then used in a third order Taylor expansion to calculate a nonlinear prediction for the update in the model variables T, V and $\mathbf{n}$ at constant

μ. The Taylor expansion is given by:

$$\begin{aligned}\Delta T = {} & X_{SS}\Delta S + X_{VS}\Delta V + \tfrac{1}{2!}X_{SSS}(\Delta S)^2 \\ & + \tfrac{1}{2!}X_{VVS}(\Delta V)^2 + X_{SSV}\Delta S\Delta V \\ & + \tfrac{1}{3!}X_{SSSS}(\Delta S)^3 + \tfrac{1}{2}X_{SSSV}(\Delta S)^2\Delta V \\ & + \tfrac{1}{2}X_{SVVS}\Delta S(\Delta V) + \tfrac{1}{3!}X_{SVVV}(\Delta V)^3 \end{aligned} \tag{3.26}$$

$$\begin{aligned}\Delta \mathbf{n} = {} & -X_{S\mu}\Delta S - X_{V\mu}\Delta V - \tfrac{1}{2!}X_{SS\mu}(\Delta S)^2 \\ & + \tfrac{1}{2!}X_{VVS}(\Delta V)^2 + X_{SSV}\Delta S\Delta V \\ & - \tfrac{1}{3!}X_{SSS\mu}(\Delta S)^3 - \tfrac{1}{2}X_{SS\mu V}(\Delta S)^2\Delta V \\ & - \tfrac{1}{2}X_{SVV\mu}\Delta S(\Delta V) - \tfrac{1}{3!}X_{VVV\mu}(\Delta V)^3 \end{aligned} \tag{3.27}$$

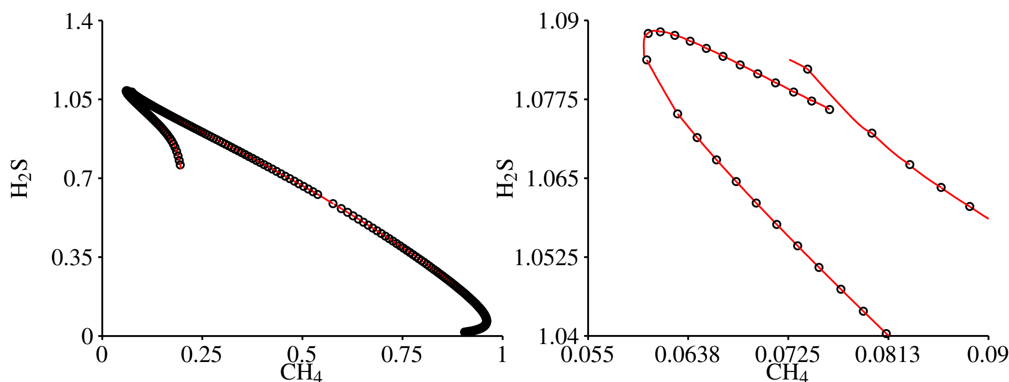


Figure 3.10: Projection of mole numbers of CH_4 versus mole numbers of H_2S (circles) with prediction (lines). In the right figure the area of abrupt changes is shown in more detail. The bad prediction is due to a singular point in the energy potential used.

Figure 3.10* shows a plot of mole numbers of CH_4 against the mole numbers of H_2S. This figure shows that the prediction is in general very good, but even though the projection is otherwise smooth, the prediction experiences serious trouble at two points. The reason for this is a singularity in the Hessian matrix $U_{\mathbf{nn}}$, which is needed to calculate the derivatives of $X(S,V,\mu)$. No further investigations about this singularity were done, but it was observed that the singularities coincide with the cusp points shown in Figure 3.7. Since the curve is smooth in this area it should

*Please note that the system size is not the same for each point. The change of system size is only prominent when the predictions are bad. It can seem like there are two different branches on the figure to the right. This is due to the scaling of the system, and will disappear with a higher resolution, but so will the bad prediction.

be possible to step over the singularity by extending the previous prediction beyond the singular point. Another observation is that the prediction is not as good in the area where the projection changes rapidly, this is as expected and it is indeed a limiting factor for the step size in this area. Figure 3.8 and 3.9 show that the projection of the other variables is smooth. This support the observation that it is possible to predict beyond the singular point, and thereby avoid problems.

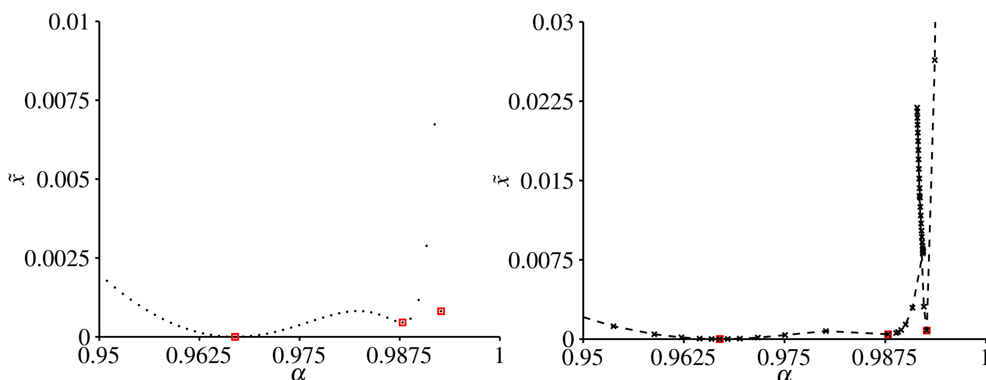
The final method, later referred to as explicit step method, combines the two methods described here. The simple method is used whenever possible, but the update in volume is at the same time calculated from the null space of Equation 3.22. It must be noted that the higher-order gradients required by the full Taylor expansion is not calculated unless necessary. If the step indicates that the update in volume is changing sign, the methods are switched. The previous value of α is then stored, and the simple method is not switched back until the value of α is beyond this point. The two methods are also switched if the simple method is not able to converge. The state is then reset to the previously converged state before the methods are switched. When the Taylor expansion in Equations 3.26 and 3.27 gives a prediction step which does not converge, the previous converged step is increased until the prediction converges again. If this approach is unsuccessful the method stops, and a global solution is not guaranteed. The step size should then be reduced. No control of the step size is implemented for the null space method, and therefore a constant step size controlled by the user is used.

The final results can be seen in Figure 3.11, and compared with the simple method from Figure 3.6(b). In these figures only the part where the tangent plane distance is close to zero is shown.

It is by combining these two methods possible to trace out the curve, and locate the minima with larger steps than before. Most of the calculations are needed close to the swallowtail catastrophe. This section is very narrow in figure 3.11, and can be better seen in Figure 3.7. The extra points in the swallowtail is due to a constant step size for the null space method used to calculate that region.

3.3 An Additional Example

In the previous section, a method for locating minima on a manifold with constant chemical potentials was developed. This method is designed for multi-component systems. The current example, however, has only two components, and it is therefore of interest to test the method also for a three-component system. The catastrophe in Figure 3.7 also diverted much attention, and it would be of interest to



(a) α–$\tilde{x}$ projection with a simple stepping method.

(b) α–$\tilde{x}$ projection where the swallowtail catastrophe is solved by a null space method.

Figure 3.11: Comparison of the intermediate and the final method for projecting the tangent plane distance $\tilde{x}$ versus α for the system CH_4–H_2S

test the method on a system with no such peculiarities. The system CH_4, nC_6, H_2S was used by Michelsen (1986); Kohse and Heidemann (1993) and Brendsdal (1999) for critical point calculations, and for studying multiphase phenomena. The starting point for the calculations done here is a converged four phase split. This should give four minima, and all of them should have a tangent plane distance within the convergence specification. As can be seen from Figures 3.12-3.14, the results are as expected. The projections of V–p and S–T give in this case the same information as the α,$\tilde{x}$ projection shown in Figure 3.12.

3.4 The Use of ODE Solvers

So far the problem has been solved by explicit steps in volume and using the fact that the volume is a linear function of α. The rest of the variables were found by solving a set of algebraic equations using Newton iteration. Even though the phase stability is a problem with special properties, it is tempting to try a more general solution approach. The tracing procedure of the tangent plane distance can easily be rewritten as a system of ordinary differential equations (ODE). This means that it is possible to integrate along the constant μ-manifold. Since the model implementation used here provides more information than a general function does, it was decided to implement the ODE solvers by hand. It is for instance possible to calculate the exact global error in each step. This is defined as the distance

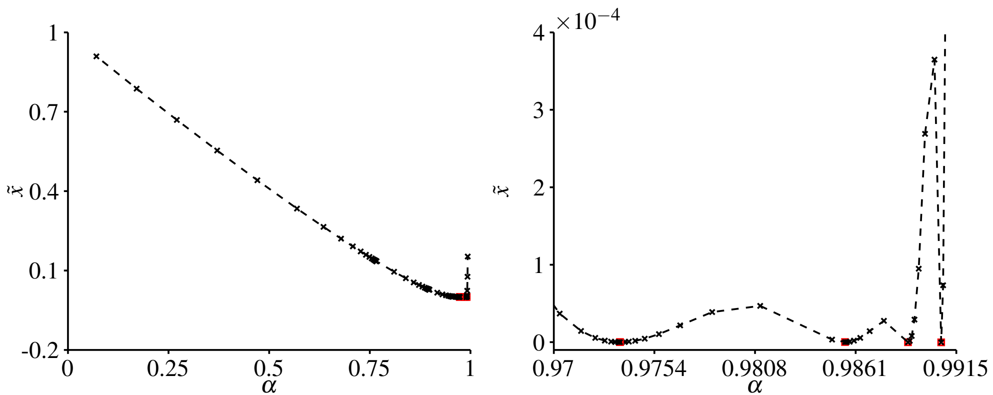


Figure 3.12: Projection of the tangent plane distance $\tilde{x}$ versus α for the system CH_4–nC_6–H_2S. The left figure shows the whole curve, while the right figure only shows the area where the minima occur.

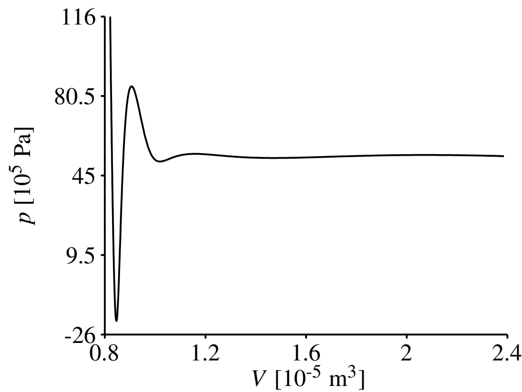


Figure 3.13: Projection of p–V at constant S and μ for the system CH_4–nC_6–H_2S.

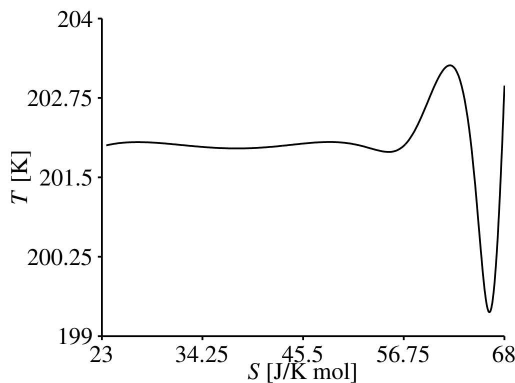


Figure 3.14: Projection of T–S at constant V and μ for the system CH_4–nC_6–H_2S

from the constant μ-manifold and μ is calculated in each step anyway. Methods with built-in error estimates are not be able to take advantage of this information, and in order to estimate the error the solver must do more function evaluations in addition to solving the problem. Since only the minima of the curve are of interest, it should be possible to solve the problem with less accuracy when the tangent plane distance is far from zero. The gradient and the Hessian are also calculated at each step. This information can, as for the method in the previous sections, be used to predict the minima.

The following two ODE solvers were implemented and tested out: Fourth order Runge-Kutta (RK4) and Runge-Kutta-Fehlberg (RKF). As will be explained, a set of algebraic equations must be solved for each integrator step, and since the integrators are not meant to be general purpose tools the algebraic equation are solved within the integrator*. The two-component system described in Section 3.1 proved to be a lot more complicated than the three component system described in the previous section. It was therefore decided to use the three-component system as a test case for developing the method. The ODE solvers also need initial values. It was therefore chosen to start at the same converged four phase equilibrium as described in the previous section. The starting point is used to establish α and the chemical potentials, where α is the integration variable of the problem. The constant chemical potential manifold constrains the problem has now the residual:

$$\Delta x(\alpha, \mathbf{a}) = \begin{pmatrix} -S(\mathbf{a}) + S(\alpha) \\ -V(\mathbf{a}) + V(\alpha) \\ \mu(\mathbf{a}) - \mu^\circ \end{pmatrix}$$

Here, $\mathbf{a}$ means the Helmholtz variables T,V and $\mathbf{n}$. The solution should of course stay as close to the $\Delta\mu = 0$ manifold as possible. The volume is explicit in $\mathbf{a}$, but there will be deviations in all the other variables, that is T, N_1, N_2 and N_3. This is compensated for by taking small steps to keep close to the manifold, and by defining a set of equations such that the solver makes a move close to the tangent of the manifold. The integration is based on a Newton step using the current state $\mathbf{a}$ as a starting point. By integrating $\mathbf{g}$† along the α–axis we get:

$$\mathbf{G}(\mathbf{a})\dot{\mathbf{g}} = -\mathbf{c}(\mathbf{a}_\circ) - \lambda \Delta x(\alpha, \mathbf{a}) \tag{3.28}$$

Here $\mathbf{c} = \partial \Delta x / \partial \alpha = C(|T_\circ|^{-1}, |p_\circ|^{-1}, 0, 0, \ldots)^T$, where $\mathbf{c}$ maps $\dot{\mathbf{g}}$‡ onto the tangent plane of the manifold, while $\lambda \Delta x$ corrects for nonlinearities in the form of an accumulated drift. Note that $C = |T_\circ| S + |p_\circ| V$, which is identical to denominator of α.

*The differential and algebraic equations are not solved simultaneously

†Gibbs energy canonical variables, $(T, p, \mathbf{n})$

‡Time derivative of $\mathbf{g}$

Since the thermodynamic model used is explicit in T, V, $\mathbf{n}$ the differential equations need to be written as a vector function of the Helmholtz variables. Typically, we let $\dot{\mathbf{a}} = \mathbf{J}(\mathbf{a})\dot{\mathbf{g}}$, where $\mathbf{J}(\mathbf{a})$ is the Jacobian as:

$$J \equiv a_g = g_a^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{-A_{V,T}}{A_{V,V}} & \frac{1}{A_{V,V}} & \frac{-A_{V,\mathbf{n}}}{A_{V,V}} \\ 0 & 0 & 0 \end{pmatrix} \tag{3.29}$$

Combining Equation 3.28 and 3.29, yields:

$$\dot{a} = J(a)X(a)[c(a_\circ) + \lambda \Delta x(\alpha, a)] \tag{3.30}$$

Equation 3.30 shows the ODE that is integrated on the constant μ-manifold. The next two sections deal with the actual integration routines.

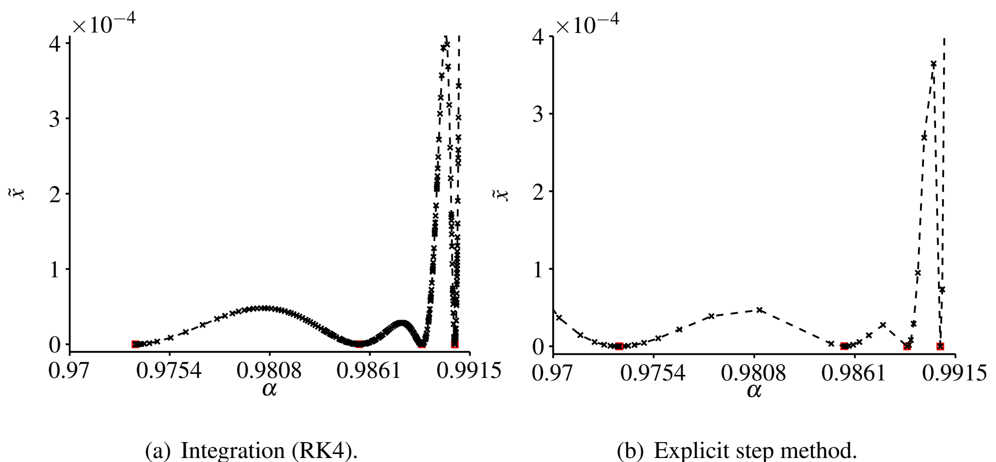
3.4.1 Fourth Order Runge-Kutta (RK4)

The differential equations were first tested by Euler's method before the actual implementation of the integrators started. Indeed, Euler's method is able to give a similar curve to the approach shown in Section 3.3. The goal is (as before) to follow the manifold and to locate all the minima of the function in Equation 3.8. In order to do this, several different ways of controlling the step size were implemented. The global criteria of constant chemical potentials can be used to make sure that the solution is close to the manifold at each integration step. As with the explicit step method a Taylor expansion is used at each point to predict the minimum. In Section 3.3 the minimum was solved as soon as it was predicted, but as explained later, a different approach of reducing the step size was tried for RK4. First, RK4 is implemented with no control of the step size. As expected this is rather slow and inaccurate compared with the explicit step method. As a first attempt to control the step size, the same Taylor expansion as explained in Section 3.2 is used to predict a minimum. Instead of solving for the exact minimum, the step size is reduced to a fixed value. When no minima is predicted, the step size is increased. This gives stability problems because when the step size grows, the deviation from the manifold becomes so large that the variables get outside the bounds of the thermodynamic model. The RK4 method has therefore an internal step size control which reduces the step size until the deviation from the μ-manifold is within a prescribed tolerance.

The accuracy is not so important far away from the minima, and it was therefore attempted to implement an adaptive tolerance control. This was not successful because the change of the tolerance during integration sometimes caused stability

problems in the integrator. It appeared that the integrator was very sensitive to changes in the tolerance while integrating. It is possible to change the tolerance at the start of the integration, but not during the integration. After several attempts to improve the method, it was concluded that the reduction of the tolerance to keep the method close to the minimum had little or no effect on the solution. Since the effect was minimal, and the method became unstable, the adaptive change of the tolerance was removed. No further investigations regarding the tolerance were made.

The way the RK4 method is implemented here makes it a hybrid solution between the explicit step method in Section 3.2 and an ODE solver*. As can be seen from Figure 3.15 the results are comparable, but the explicit step method requires fewer function evaluations to trace out the curve. It should, however, be noted that considerably more effort were put into improving the explicit step method than tuning the RK4 solver. It should also be noted that for the RK4 method there is no check to see that the S–V relationship is constant, as this is explicitly assumed by the integrator. The predicted value of α was larger than one at the end of integration, indicating that this criterion is not fulfilled. No further investigations were done, but from Figure 3.15 no significant deviations can be seen.



(a) Integration (RK4).

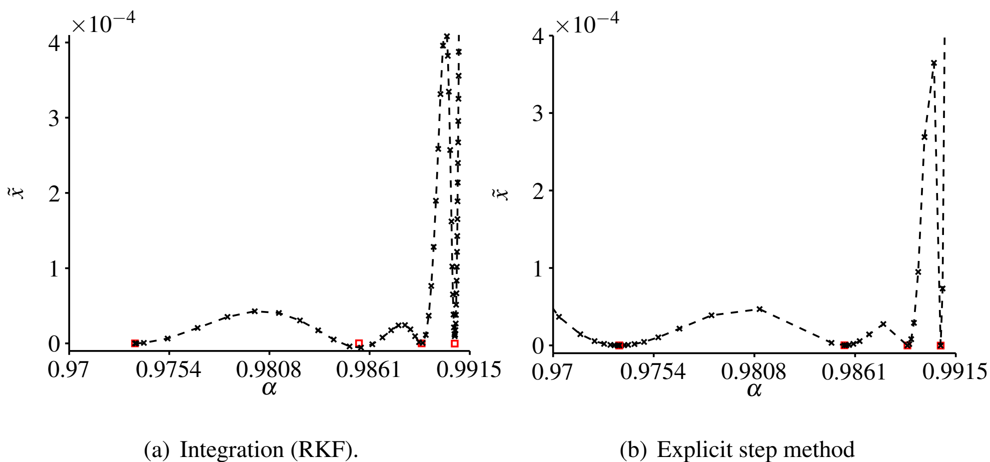
(b) Explicit step method.

Figure 3.15: Comparison of integration and iteration for the projection of the tangent plane distance $\tilde{x}$ versus α for the system CH_4–nC_6–H_2S. The crosses show each point calculated. The squares show the converged minima. The dashed lines show the trend based on the points.

*This is not a DAE solver, as the differential and algebraic equations are solved separately

3.4.2 Runge-Kutta-Fehlberg (RKF)

A step size control was implemented for the RK4 method based upon the knowledge of the problem and the fact that only parts of the curve are of interest. The integration is comparable to the explicit step method, but the RK4 solver is no longer a pure ODE solver. Without any step size control the RK4 method is, as expected, either very slow or inaccurate. The natural alternative would of course be to implement an integration method with built-in control of the step size. The RKF method is a simple ODE solver with built-in step control. The RKF method will naturally not be able to take advantage of the information used to construct the step size control for the RK4 method and for the explicit step method. This means that the implementation will not be able to increase the resolution close to the minima*. The RKF method needs more function evaluations per step to find an error estimate, which is used to control the step size. However, this is based on local information only, and no global error is predicted. The execution time of the RKF method depends therefore on the tolerance of the local error. When the tolerance is set to 10^{-2} the runtime is about the same as for the RK4 method, and, as can be seen from Figure 3.16, the accuracy is good.



(a) Integration (RKF).

(b) Explicit step method

Figure 3.16: Comparison of integration and iteration for the projection of the tangent plane distance $\tilde{x}$ versus α for the system CH_4–nC_6–H_2S. The crosses show each point calculated. The squares show the converged minima. The dashed lines show the trend based on the points.

*It will still be possible to detect a minimum outside the method, and externally change the step size, or reduce the step size. This was not done in this case

3.5 Using DAE Solvers in Matlab

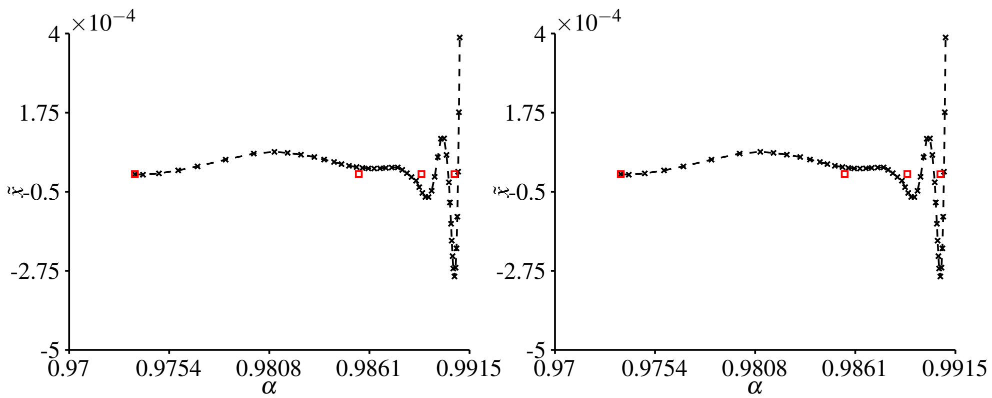
A third way of tracing out the tangent plane distance as a function of α, is to use a differential algebraic equation (DAE) solver. Matlab has several DAE solvers, and the problem is easily reformulated such that these can be used. The Matlab DAE solvers need an independent differential variable called time, and a set of initial conditions in order to solve the problem. The current problem, as shown in Equation 3.30, has no physical time but the α variable can be used instead. Since both S and V are explicit variables in α, but they can be solved as differential equations. V can also be solved as the only differential variable. This provides the alternative of either one or two differential equations in the DAE. The algebraic equations make sure the integration stays close to the μ-manifold. Both ode23t and ode15s have been tried and, as can be seen from Figure 3.17, they give similar results for the default accuracy, but the results are not as good as the other methods presented in this chapter. In order to get satisfactory results the tolerance had to be decreased, but with higher accuracy the ode23t was not able to solve the problem.

It should be noted that the denominator of α in Equation 3.4 is assumed constant during the integration. This is not entirely true in the DAE solver, which makes the termination of the integration a bit tricky, since the value of α used by the DAE solver is not exactly the same as the real one. This affects the overall runtime of the solver but not the numerical results, since only the real α is used in the calculations.

3.6 Conclusion

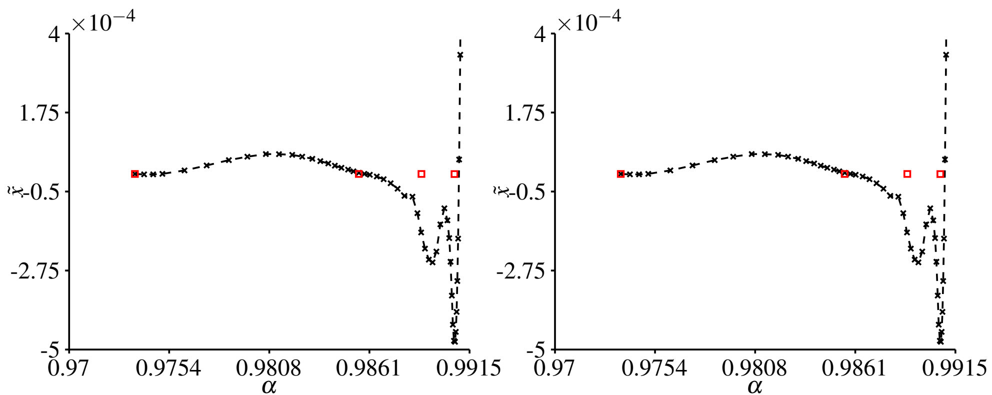
Three approaches to the phase stability test have been tested, and in all the approaches the energy potential $X(S,V,\mu)$ was used. This energy potential was chosen because the chemical potentials are kept constant during the calculations, leaving only two free variables regardless of the number of components in the system. Unfortunately, there exist no thermodynamic models using these coordinates. The Soave-Redlich-Kwong equation of state was used in this work, and the Legendre transform was used to achieve the desired model formulation. The update of the variables must still be defined in the SRK coordinates, and the calculated updates in S and μ, are therefore converted in a Jacobian transformation to similar updates in T and $\mathbf{n}$.

In phase equilibrium calculations it is always a question whether the phases are stable, or new phases will form. When instabilities occur it is not easy to figure



(a) ode23t with one differential equation

(b) ode23t with two differential equations



(c) ode15s with one differential equation

(d) ode15s with two differential equations

Figure 3.17: Comparison of the use of different Matlab DAE solvers for projecting the tangent plane distance $\tilde{x}$ versus α for the system CH_4–nC_6–H_2S. The crosses show each point calculated. The squares show the converged minima. The dashed lines show the trend based on the points.

out how many of the phases will be present in the stable system, and which ones are unstable. In this chapter a method for checking that the phase assembly is globally stable has been developed, and the two-component system CH_4–H_2S has been studied in details. The same system is known from the literature, and some results were reproduced to make sure the thermodynamic model had similar parameters. All calculations made here are consistent with reported works.

At equilibrium all the intensive properties T, p and μ are the same in all phases, and the stability check developed in this work utilises this fact. By constraining all but two intensive variables to their values at the starting point, it is possible to easily explore the thermodynamic space. The manifold defined by the constant μ's is in this case a constraint to the stability check, and the two extensive variables left for the system are used to span the thermodynamic space. For a two-component system, this can be done by varying the mole fractions from zero to one at constant T and p. In the approaches presented here a similar normalization is used for S and V at constant μ. The crucial point is that the last approach is amenable to multi-component systems, while the first one is not. The reason for this is that adding a new component will not affect the number of extensive variables when all the μ's are kept constant. The normalization of S and V is achieved by defining a linear relationship between S and V, equal to $\alpha = \frac{|T_0|S}{|T_0|S+|p_0|V}$. This gives a variable $\alpha \in \langle 0, 1 \rangle$, as for the mole fractions. For the two-component system CH_4–H_2S, both normalization approaches give equivalent results, as both show three minima and one swallow tail catastrophe.

The variable α was used in several different methods for finding the tangent plane distance minima in the system. The study incorporated one explicit step method, two different ODE solvers, and two different DAE solvers. The explicit step method gave the most control, but the two ODE solvers RK4 and RKF, which were adapted to this problem, also worked satisfactory. Without further investigations it is not possible to tell which approach is the best. From the figures presented here it looks like the explicit step method is better, but on the other hand considerable more time were spent on improving this approach. The two Matlab DAE solvers ode15s and ode23s also solved the problem, while ode23s were not able to solve the problem with satisfactory accuracy. The DAE solvers are Matlab tools, and no problem specific knowledge could be implemented to help solving the problem. However, the implementation of a tailor-made DAE solver is probably not worth the effort since the performance of the Matlab DAE solvers were considerably lower than all other approaches.

The explicit step method presented here combines two different methods. The simplest method makes one explicit step forward in α. At each step Newton's

method is used to stay close to the manifold. If this fails the step is rejected and a more advanced, and thereby slower method is used. The current suggestion is to use a third-order Taylor expansion in T and $\mathbf{n}$ to predict an update which is close to the manifold. If the Taylor expansion moves backwards in α compared to what the simple explicit method would do, the method switches to the Taylor expansion method until the value of α has passed the point where the simple method failed. Otherwise the simple method is attempted in the next step also. This proved to work satisfactory for the two test cases. All minima were detected, and the method was able to explore the thermodynamic space.

3.6.1 Further work

The stability check discussed in this chapter shows only what is possible to calculate for a multi-component system. The method is only used to create a graphical presentation of the solution, and is so far not useful to other algorithms. To find the stable phases of a system the method must be combined with a generic flash algorithm, which calculates the correct number of phases.

It is likely that the ODE solvers could be developed further, and the explicit step method should definitely be properly compared with the ODE solvers. It is, however, not recommended to implement generic numerical methods, because a problem specific knowledge can be utilised to a large extent in order to significantly improve the performance. If the numerical method is too complex, it is likely that it will be more difficult to implement any prior problem knowledge.

Certainly, the explicit step method demonstrated here, has room for improvements. The step size control is more or less left to the user, which in some cases is quite adequate, but a semi-automatic way of handling this problem should be found. In particular the Taylor expansion uses a fixed step size, and a method for adaptive step control should be sought.

Chapter 4

Criticality

In Part II, a system for generating tailor-made thermodynamics libraries is described. This description involves both the automatic generation of higher-order gradients and the automatic Legendre transform (also with gradients). In this chapter it is verified that the calculations of the exported code yield reasonable results*, and at the same time the use of higher-order gradients of the Legendre transformed energy potential is demonstrated. For the purpose of demonstration it was decided to explore the fluid region close to critical and tri-critical points as this requires higher-order gradients, and is indeed a nontrivial task. The ternary system CH_4–nC_6–H_2S including all three binary subsystems were chosen as test cases. These systems have been studied on several occasions by Michelsen (1986) and Kohse and Heidemann (1993), and the calculations presented here will mostly be a reproduction of previous results, but with a different approach.

The main focus is to locate critical and tri-critical points. For all calculations, the Soave–Redlich–Kwong (SRK) equation of state (Soave, 1972) is used, which is the same model as used in the referred works. The converged solutions do not match perfectly, but this is probably due to differences in the model parameters, which for the published works are not precisely known. To locate the tri-critical point Michelsen (1986) and Kohse and Heidemann (1993) used a minimization of the tangent plane distance expanded in a Taylor series. In the current work higher-order gradients and Legendre transforms are available, and the critical criteria out-

*The automatically generated code had been tested beforehand, and all tests indicated that the code calculated the correct result. With such a complex code, it is not possible, or at least not desirable, to test the code by inspection. Therefore numerical tests must be applied. One can never be 100 % certain that everything is covered in numeric tests. It is therefore important to test the code on actual problems with known results.

lined by Reid and Beegle (1977) can therefore be utilised directly. Their criteria depend on the specific ordering of the variables, and different orderings give different, albeit equivalent criteria. Common to all the criteria is that the Legendre transform is used to define an energy potential with only two extensive variables. It should be noted that Reid and Beegle (1977) used determinants to calculate the properties of the Legendre transform, while arrays of partial derivative are used in this work. To define the critical conditions, a partial derivative of the energy potential with respect to one of the two extensive variables is calculated, keeping all the intensive variables constant. This gives rise to several alternative, but as said, equivalent criteria. In the present work it was chosen to use an energy potential with entropy (S), volume (V) and chemical potentials (μ) as the free variables. This function has no common name, and will for the time being be referred to as $X(S, V, \mu)$. As will be explained later, this function makes it possible to define critical criteria which are independent of the number of components.

In addition to the critical point calculations, a Taylor expansion at the critical point of the two-component system CH_4–H_2S was used to predict equilibrium points along the two–phase boundary. Traditionally, the two–phase boundary is located by starting at a point well away from the critical point, and then attempting to close the phase curve at the critical point. With the traditional approach numerical problems can occur close to the critical point, as shown by Michelsen (1984). This is because the pertinent matrices will be singular at the critical point. It is therefore interesting to note that we are able to calculate the phase boundary by Taylor expansion starting from a point which is mathematically singular. The success lies in the understanding that the singularity is of limited order. Hence, a Taylor expansion of sufficient order is well defined even at the critical point.

4.1 Critical Lines in Binary Systems

It is absurd to start looking for a tri-critical point without being sure that the automatically generated code is correct, and without knowing that the theory leads to a solution. It was therefore decided to develop things gradually, and gain some insight along the way. The natural place to start is to find the one-component critical point first. The answer is here dictated by the parameters of the model, but an erroneous model will most likely not converge to the correct solution. The iteration table for converging the pure component critical point for CH_4 is shown in Table 4.1. This table shows that the convergence of Newton's method close to the solution (iterations 4-6) is of second order. For starting values sufficiently close to the solution, all the pure-component critical points were found (CH_4, nC_6, H_2S).

Table 4.1: Iteration sequence to pure component critical point for CH_4.

k	$\frac{T-T_c}{T_c}$	$\frac{V-V_c}{V_c}$
0	1.7207e-02	-1.1791e-01
1	6.5276e-03	-5.1454e-02
2	1.5375e-03	-1.1933e-02
3	9.9242e-05	-7.4296e-04
4	4.1242e-07	-3.0239e-06
5	6.9245e-12	-5.0354e-11
6	-8.7390e-16	1.2084e-16

It is slightly more difficult to trace out the critical line from one pure-component critical point to the other, as there might not be a continuous solution. According to van Konynenburg and Scott (1980), many binary systems have continuous critical lines. These belongs to Class 1 in their phenomenology characterisation. Systems with discontinuous critical lines are called Class 2. In the three component system investigated here, three binary possibilities exist. One system is of Class 1 (nC_6–H_2S), and two of the systems are of Class 2 (CH_4–H_2S and CH_4–nC_6).

All the critical lines were traced with the same algorithm, as described below. First and foremost, the critical point needs to be located at each point on the curve. Newton's method was used to converge to an arbitrary critical point. The general criteria for an n'th-order critical point given by Reid and Beegle (1977), are also repeated in Chapter 2. The criteria state that an energy potential with two, and exactly two, extensive variables is sufficient to formulate the critical condition for a given system. In a binary system this means that two of the variables must be intensive, while the other two variables are extensive. For practical reasons it is undesirable to split the mole numbers into disjoint sets*, which means there are only two possible energy potentials available, namely Gibbs energy $G(T, p, \mathbf{n})$, and the unnamed function $X(S, V, \mu)$. The critical point criteria from these energy potentials are equivalent, and the particular choice of the potential is therefore only of practical importance.

In Section 4.2 on page 56, the criteria for a tri-critical point are repeated. A tri-critical point can exist in systems with three or more components. This means that Gibbs energy cannot be used to formulate the criteria for the tri-critical point, since the mole numbers of the three components give rise to three extensive variables. The energy potential $X(S, V, \mu)$ has no such problems, since the number of components in the system is reflected through the intensive μs only. To be able to

*This will significantly increase the implementation effort for the model

handle systems with more than two components, it was therefore decided to use X to formulate the critical criteria. Still, there is a nontrivial choice of variables to formulate the critical condition because either S or V can be used as the free variable of the critical condition. In this work V was chosen, but a choice of S would have been equivalent. However, the equation of state used is explicit in V, which makes this variable the desired choice. The criteria for the critical point of an n-component system are given below:

$$\begin{aligned} X_{VV} &= -(\partial p/\partial V)_{S,\mu} = 0 \\ X_{VVV} &= -(\partial^2 p/\partial V^2)_{S,\mu} = 0 \end{aligned} \tag{4.1}$$

In addition, the third derivative of pressure with respect to volume must be positive if the critical point shall be a stable one. Here, the definitions $X_V \hat{=} (\frac{\partial X}{\partial V})_{S,\mu} \hat{=} -p$, $X_{VV} \hat{=} (\frac{\partial^2 X}{\partial V \partial V})_{S,\mu}$, etc. are used. This gives the following Newton iteration for solving the critical point conditions:

$$\begin{pmatrix} X_{VVS} & X_{VVV} \\ X_{VVVS} & X_{VVVV} \end{pmatrix} \begin{pmatrix} \delta S \\ \delta V \end{pmatrix} = \begin{pmatrix} X_{VV} \\ X_{VVV} \end{pmatrix} \tag{4.2}$$

As a thermodynamic exercise, and also as an additional test of the automatically generated code, the Euler theorem of homogeneous functions (see Callen (1985) and Modell and Reid (1983)) is used to formulate another critical point criteria, which will be compared to the one already stated. As shown by Haug-Warberg (2006) the following can be stated:

$$\begin{pmatrix} X_{SS} & X_{VS} \\ X_{SV} & X_{VV} \end{pmatrix} \begin{pmatrix} S \\ V \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This is always true, but since X_{VV} is zero at the critical point, and V and S cannot be zero, the rest of the matrix must also be zero. Furthermore, the third-order derivative must satisfy the following relation:

$$\begin{pmatrix} X_{SSS} & X_{VSS} \\ X_{SVS} & X_{VVS} \end{pmatrix} S + \begin{pmatrix} X_{SSV} & X_{VSV} \\ X_{SVV} & X_{VVV} \end{pmatrix} V = - \begin{pmatrix} X_{SS} & X_{VS} \\ X_{SV} & X_{VV} \end{pmatrix}$$

This means that X_{VVS} will approach zero as X_{VV} approaches zero. This can then be used as an alternative condition at the critical point. The same argument applies to X_{VVV}:

$$\begin{aligned} X_{VVS} &= -(\partial^2 p/\partial V \partial S)_{\mu} = 0 \\ X_{VVV} &= -(\partial^2 p/\partial V^2)_{S,\mu} = 0 \end{aligned} \tag{4.3}$$

According to 4.3 the matrix on the left hand side of Equation 4.2 is singular at the critical point. This is purely a numerical problem, which makes convergence more difficult. The numerical problems can be avoided by iterating until the norm starts increasing. This approach can of course make the iteration stop far away from the true solution to the problem. However, while doing these calculations this problem was not encountered.

As mentioned earlier, the Soave–Redlich–Kwong equation of state is used for all the practical calculations. This is a thermodynamic model with state variables T, V and $\mathbf{n}$. In Criteria 4.2 and 4.3 it is stated that the chemical potentials are constant. This is not possible to achieve directly since μ are dependent variables, and therefore the system of equations must be extended to keep the μs constant. This will then give the Newton iteration shown below.

$$\begin{pmatrix} X_{VVS} & X_{VVV} & X_{VV\mu} \\ X_{VVVS} & X_{VVVV} & X_{VVV\mu} \\ X_{VV\mu S} & X_{VV\mu V} & X_{VV\mu\mu} \end{pmatrix} \begin{pmatrix} \delta S \\ \delta V \\ \delta\mu \end{pmatrix} = - \begin{pmatrix} X_{VV} \\ X_{VVV} \\ 0 \end{pmatrix} \tag{4.4}$$

The equations added in 4.4 form a null space which will give an update in the variables with a minimal change in the chemical potentials. This is required since the model is not explicit in μ, and a change in the other variables will also change μ. The model is formulated in T, V and $\mathbf{n}$, while the update from the Newton iteration is in S, V and μ. The update must therefore be mapped back to the original coordinates. This is done by the Jacobian shown below.

$$\frac{\partial(S,V,\mu)}{\partial(T,V,\mathbf{n})} = \begin{pmatrix} A_{TT} & A_{TV} & A_{T\mathbf{n}} \\ 0 & 1 & 0 \\ A_{\mathbf{n}T} & A_{\mathbf{n}V} & A_{\mathbf{nn}} \end{pmatrix} \tag{4.5}$$

From solving the linearised Equation (4.6), the approximate update is found, and the procedure will converge to a critical point provided the starting point is sufficiently close to the solution. It is, however, assumed that one iteration is sufficient to map from one variable set to another. This is of course not true, and therefore the update will be a little different from the exact mapping. No convergence problems were encountered, however, which supports the assumption that one iteration is generally sufficient to map from S, V, μ to T, V, $\mathbf{n}$. It does at least no change the convergence region very much.

$$\begin{pmatrix} A_{TT} & A_{TV} & A_{T\mathbf{n}} \\ 0 & 1 & 0 \\ A_{\mathbf{n}T} & A_{\mathbf{n}V} & A_{\mathbf{nn}} \end{pmatrix} \begin{pmatrix} \delta T \\ \delta V \\ \delta\mathbf{n} \end{pmatrix} = - \begin{pmatrix} \delta S \\ \delta V \\ \delta\mu \end{pmatrix} \tag{4.6}$$

The convergence criteria for the Newton method is defined by comparing the norm of the right hand side of Equation 4.4 with the norm of the previous step. This means that the method will proceed to the solution as close as possible. The convergence criteria assume of course that the method converge to the correct solution. The actual convergence is inspected by checking the size of the norm on the outside of the iteration loop. The advantage of this approach is that it will not result in an infinite loop, and the resolution of the result is as good as the Newton method is capable of. This is ensured because the Newton method will approach the solution monotonically if the method converges. The traditional convergence criterion is to stop when the update approaches zero, but the problem is to define what zero means in a numerical context.

When tracing out the critical line a new critical point at different composition is sought once a critical point has been converged. From the last converged point it is possible to predict in which direction the critical line moves. This is done by removing one row from Equation 4.4, and calculating the null space of the matrix. In order to avoid poor scaling of the system*, one of the remaining rows of the matrix on the left hand side is replaced by $[S,V,0,0]$. This means that the prediction will be in an orthogonal direction to the vector $[S,V]$, which means the size of the system is not affected seriously. In order to avoid cycling between two solutions it must be ensured that subsequent predictions point in the same main direction. This is done by comparing the direction of the previous step with the direction of the new prediction, and if the inner product between the predictions has a negative sign, the direction of the last prediction vector is flipped.

The length of the null space vector is always unity in Matlab. This means that the prediction needs to be scaled in order to produce a decent update. Pragmatically, it is ensured that the change in any variable is smaller than a given percentage. How large this step can be is problem dependent. For the systems studied here, the maximum variable change varied from one to ten percent, without experiencing stability problems.

4.1.1 Results

Figure 4.1 shows that the system CH_4–H_2S has a discontinuous projection in the intensive T–p plot. Figure 4.1 is equivalent to Figure 12 of Heidemann and Khalil (1980). The endpoint of the cusp lies on the stability limit of the critical line.

*By multiplying the extensive variables with the same value, the system is scaled, but the intensive properties remain the same. This means that the procedure will converge to the same solution. This is due to Euler's theorem(Modell and Reid, 1983).

The line of decreasing pressure and temperature represents unstable critical points. This means that the fourth-order derivative of X with respect to the volume is negative for the converged points, and according to Heidemann and Khalil (1980) this is a three phase liquid–liquid–gas line. The cusp, or Riemann–Hugoniot catastrophe, see Thom (1975), is due to a folding in the multidimensional space of which this plot is a projection. Interestingly, the cusp point is coinciding with a maximum in both T and p. This can be seen from Figure 4.3 where the point is indicated with a circle.

The cusp shown in the intensive projection of Figure 4.1 does not appear in an extensive projection of the critical line. It should be noted that because the system size is not constant while tracking the curve, it is difficult to give a clear picture of the evolution of the extensive properties. It was therefore decided to scale the extensive variables with the total number of moles of the system at each point. As can be seen from Figure 4.2, this gives two smooth continuous lines.

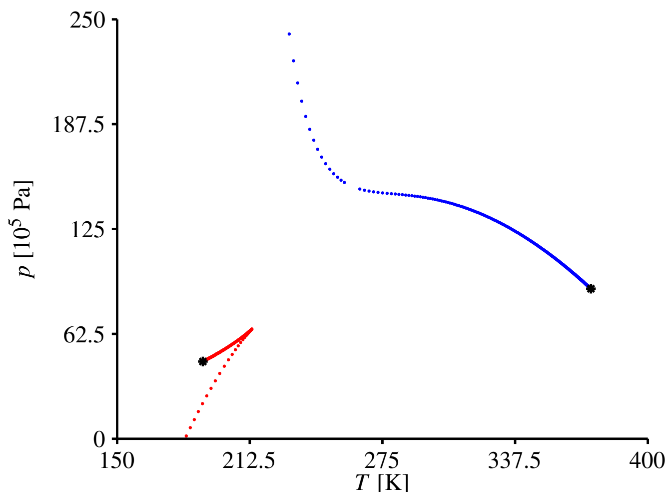


Figure 4.1: Intensive projection of critical lines for the system CH_4–H_2S. The one-component critical points are indicated by an asterisk.

Figure 4.4–4.6 show similar plots for the two remaining binary systems. The scaled extensive projections are always smooth, while the system CH_4–nC_6 also shows a cusp in the T–p projection (Figure 4.4). In Figures 4.6(a) and 4.6(b) only one line is shown. This is because the critical line goes continuously from the one pure component critical point to the other. The trace of the lines is started in both the endpoints, but the two lines are coinciding, and therefore only one line is visible.

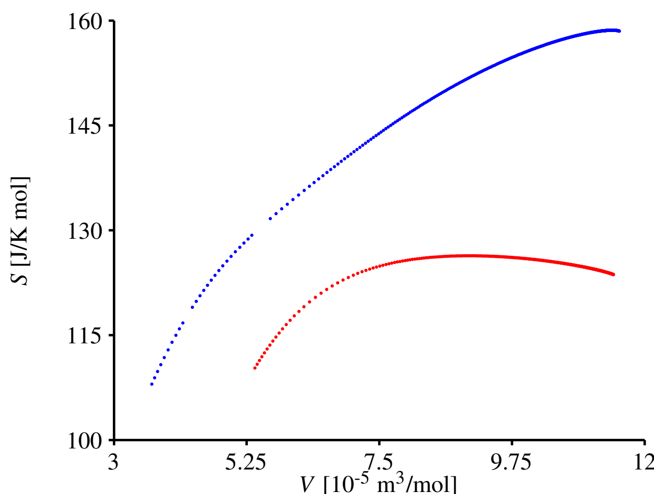


Figure 4.2: Molar projection of critical lines for the system CH_4–H_2S.

4.2 The Tri-critical Point

At the tri-critical point three phases become indistinguishable. According to Gibbs phase rule, there must be at least three components to give sufficient degrees of freedom for a tri-critical point, see Knobler and Scott (1984). This can be seen from the phase rule at the critical point where n phases become identical; $F = C+3-P-n$. When $P=3$ and $n=3$, the number of components has to be at least three. In the previous section, only two–component systems were investigated. This means that one extra component must be added in order to have a system with a possible tri-critical point. The same energy potential as for the two-component critical line calculations is used, but the criteria for the critical conditions need to be extended. The criteria for the tri-critical point can be formulated in the energy potential X, as shown below. Note that Equation 4.7e is needed only to make sure that the solution is a stable one.

$$X_{VV} = -(\partial p/\partial V)_{S,\mu_1,\mu_2,\mu_3} = 0 \tag{4.7a}$$

$$X_{VVV} = -(\partial^2 p/\partial V^2)_{S,\mu_1,\mu_2,\mu_3} = 0 \tag{4.7b}$$

$$X_{VVVV} = -(\partial^3 p/\partial V^3)_{S,\mu_1,\mu_2,\mu_3} = 0 \tag{4.7c}$$

$$X_{VVVVV} = -(\partial^4 p/\partial V^4)_{S,\mu_1,\mu_2,\mu_3} = 0 \tag{4.7d}$$

$$X_{VVVVVV} = -(\partial^5 p/\partial V^5)_{S,\mu_1,\mu_2,\mu_3} \geq 0 \tag{4.7e}$$

Four degrees of freedom are required to satisfy these criteria. The fifth degree of

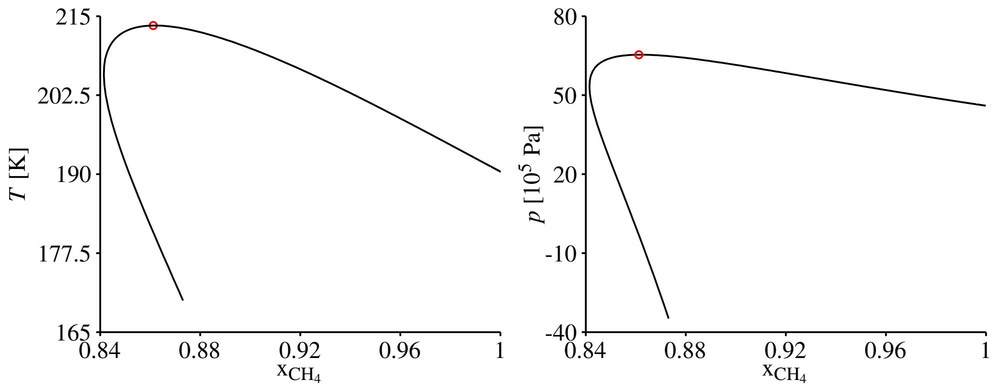


Figure 4.3: The cusp point from Figure 4.1 correspond to a maximum in T and p, here plotted against mole fraction of CH_4. The cusp point is indicated with a circle.

freedom is used to make sure that the system is not scaled during the update*. Scaling of the system can be avoided by keeping one of the extensive variables constant during the iteration. In this work it was chosen to keep the volume constant. This is the natural choice, since volume is the only variable controlled explicitly in the update. This choice gives the following Newton iteration scheme:

$$\begin{pmatrix} X_{VVSSSS} & X_{VVSSSV} & X_{VVSSS\mu} \\ 0 & 1 & 0 \\ X_{VVVSSS} & X_{VVVSSV} & X_{VVVSS\mu} \\ X_{VVVVSS} & X_{VVVVSV} & X_{VVVVS\mu} \\ X_{VVVVVS} & X_{VVVVVV} & X_{VVVVV\mu} \end{pmatrix} \begin{pmatrix} \delta S \\ \delta V \\ \delta\mu_1 \\ \delta\mu_2 \\ \delta\mu_3 \end{pmatrix} = - \begin{pmatrix} X_{VVSSS} \\ 0 \\ X_{VVVSS} \\ X_{VVVVS} \\ X_{VVVVV} \end{pmatrix} \tag{4.8}$$

Equation 4.8 use the stability criteria formulated in Equation 4.7, but in order to get derivatives of the same order in the matrix, the same type of alternative formulation as shown in Equation 4.3 is used. This does not affect the final results.

The update, and thereby the solution of the system of equations will be given in S, V, μ. Since the model is written in T, V, $\mathbf{n}$, the update must be mapped back to these variables before the actual update is made and the new state is calculated. The mapping has already been explained in Equation 4.6.

*If all the extensive properties of a state are multiplied with the same scalar value, the size of the system will be scaled equivalently. This is because the energy potential is homogeneous of order one with respect to the extensive variables

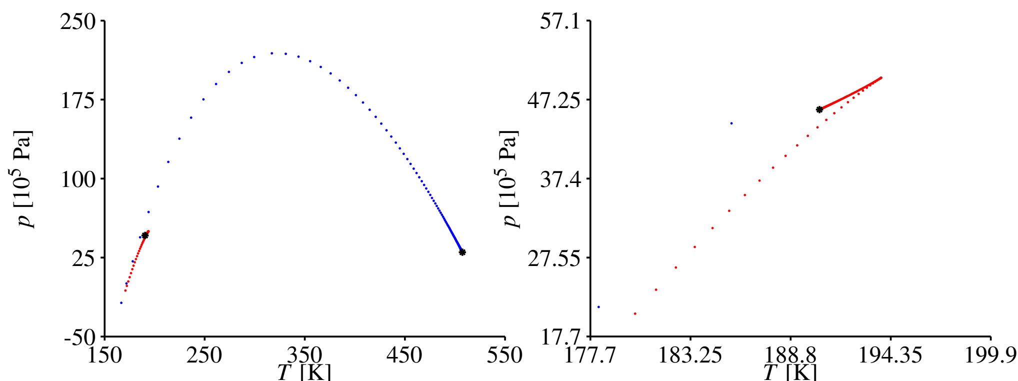


Figure 4.4: Intensive projection of critical lines for the system CH_4–nC_6. The cusp here is very narrow, and a zoomed version is also shown. The one-component critical points are indicated by an asterisk

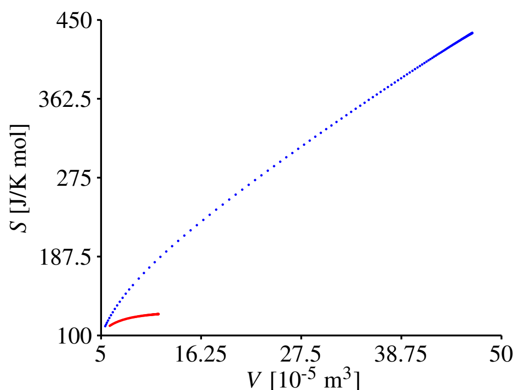
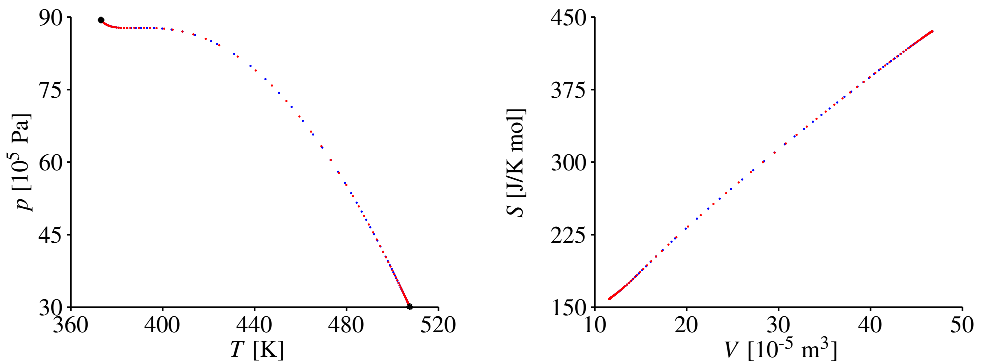


Figure 4.5: Molar projection of critical lines for the system CH_4–nC_6.

Equation 4.8 converges to the tri-critical point provided that the initial values are sufficiently close to the solution. Table 4.2 shows the convergence of the tri-critical point calculation while varying one variable at a time and keeping the other variables at their converged values. This gives an indication of the convergence region of the algorithm.

4.3 Taylor Expansion of Phase Boundaries

The critical point is an observed endpoint of the two–phase boundary, where the two phases become indistinguishable. It should therefore be possible to trace out



(a) Intensive projection of critical lines for the system nC_6–H_2S. The one-component critical points are indicated by an asterisk.

(b) Molar projection of critical lines for the system nC_6–H_2S.

Figure 4.6: Projection of critical lines for the system nC_6–H_2S.

Table 4.2: Convergence region for locating the tri-critical point of the system CH_4–nC_6–H_2S. Sown as percentage change of each variable while keeping the others constant.

	T	V	N_1	N_2	N_3
+	12	20	9	88	18
-	5	6	20	40	16

parts of the phase boundary using the critical point as a starting point. The work presented here is done as a mathematical exercise with a thermodynamic basis, and has less relevance to engineering calculations like for instance the flash calculations shown by Michelsen (1982b). The equilibrium criteria defined along the phase boundary requires that all the intensive properties must be the same in both phases. This means that the gradient of $U(S,V,\mathbf{n})$ is the same in both phases. Ordinarily, the phase boundaries are calculated at some intensive constraint, for instance at constant temperature or pressure. The predictions made by the Taylor expansion do not have such constraints, and the diagrams are therefore not easily compared with results from the literature.

The Taylor expansion of the gradient of U^* is given by the following equation, Haug-Warberg (2008):

$$g = g_\circ + U_2(\mathrm{d}x) + \frac{1}{2!}U_3(\mathrm{d}x)^2 + \frac{1}{3!}U_4(\mathrm{d}x)^3 + \mathscr{O}\left((\mathrm{d}x)^4\right) \tag{4.9}$$

*The subscript to U gives the order of differentiation.

The perturbation of the critical state has two vector components for each phase. These components are orthogonal to the critical condition, and to each other. The first component is found by locating the singular eigenvector of the Hessian matrix u_2 of internal energy, such that $U_2 q = 0$ and $q \perp z_\circ$. This component gives the same contribution to both phases, but with different sign, meaning that the mass and the energy removed from one phase will be added to the other. The other vector component is given implicitly by the Taylor expansion. The total perturbation is therefore:

$$\mathrm{d}x = +\alpha q + Nu \tag{4.10}$$

$$\mathrm{d}y = -\alpha q + Nv \tag{4.11}$$

$$\mathrm{d}x + \mathrm{d}y = N(u+v) \tag{4.12}$$

Here, N describes a plane orthogonal to both $z_\circ$ and q, which means N is given by:

$$N = \mathrm{null}\begin{pmatrix} q^{\mathrm{T}} \\ z_\circ^{\mathrm{T}} \end{pmatrix} \tag{4.13}$$

where u and v are used to select a vector in the plane N such that the Taylor expansion is satisfied. In fact, u and v are implicitly defined by the Taylor expansion. The component αq takes the perturbation away from the critical point, while Nu and Nv compensate for the fact that this is not exactly in the direction of the equilibrium manifold.

When dx and dy are inserted into 4.9 the following equations emerge:

$$g(\mathrm{d}x) = g_\circ + U_2(\alpha q + Nu) + \tfrac{1}{2!}U_3(\alpha q + Nu)^2 + \tfrac{1}{3!}U_4(\alpha q + Nu)^3 \tag{4.14}$$

$$g(\mathrm{d}y) = g_\circ + U_2(-\alpha q + Nv) + \tfrac{1}{2!}U_3(-\alpha q + Nv)^2 + \tfrac{1}{3!}U_4(-\alpha q + Nv)^3 \tag{4.15}$$

By expanding these equations, we can write:

$$\begin{aligned} g(\mathrm{d}x) = g_\circ &+ U_2\alpha q + U_2 Nu + \tfrac{U_3}{2}(\alpha q)^2 + U_3\alpha q Nu + \tfrac{U_3}{2}(Nu)^2 \\ &+ \tfrac{U_4}{6}(\alpha q)^3 + \tfrac{U_4}{2}(\alpha q)^2 Nu + \tfrac{U_4}{2}\alpha q(Nu)^2 + \tfrac{U_4}{6}(Nu)^3 \end{aligned} \tag{4.16}$$

$$\begin{aligned} g(\mathrm{d}y) = g_\circ &- U_2\alpha q + U_2 Nv + \tfrac{U_3}{2}(\alpha q)^2 - U_3\alpha q Nv + \tfrac{U_3}{2}(Nv)^2 \\ &- \tfrac{U_4}{6}(\alpha q)^3 + \tfrac{U_4}{2}(\alpha q)^2 Nv - \tfrac{U_4}{2}\alpha q(Nv)^2 + \tfrac{U_4}{6}(Nv)^3 \end{aligned} \tag{4.17}$$

The gradient must be the same in both phases in order to satisfy the equilibrium conditions. This means that $g(\mathrm{d}x) - g(\mathrm{d}y) = 0$. By using this relation and ignoring higher-order terms in u and v Equations 4.16 and 4.17 become:

$$\begin{aligned} g(\mathrm{d}x) - g(\mathrm{d}y) = {} & U_2 N(u-v) + U_3\alpha q N(u+v) \\ & + \tfrac{U_4}{2}(\alpha q)^2 N(u-v) + \tfrac{U_4}{3}(\alpha q)^3 + \mathscr{O}(u^2, v^2) \end{aligned} \tag{4.18}$$

The last equation can be solved for $u+v$ and $u-v$ which yields

$$\left[(U_2 + \tfrac{U_4}{2}(\alpha q)^2)N \quad U_3 \alpha q N\right] \begin{pmatrix} u-v \\ u+v \end{pmatrix} = -\tfrac{U_4}{3}(\alpha q)^3 \tag{4.19}$$

It is assumed that the critical phase is split in two equal parts, and the calculated perturbation is added to predict a new equilibrium a certain distance α from the critical point:

$$x = \tfrac{1}{2}z_\circ + \alpha q + \tfrac{1}{2}N[(u-v)+(u+v)] \tag{4.20}$$

$$y = \tfrac{1}{2}z_\circ - \alpha q + \tfrac{1}{2}N[(u+v)-(u-v)] \tag{4.21}$$

The last term of the prediction is unsymmetrical, which means the size of the system will be changed in the prediction. The original coordinates are used as an extensive constraint vector in the equilibrium calculations to keep the system size constant. The equilibrium is found by calculating $\min(U(x)+U(y))$ subject to $x+y=z_\circ$. The phase equilibrium code converging the $S,V,\mathbf{n}$ flash routine will not be described here.

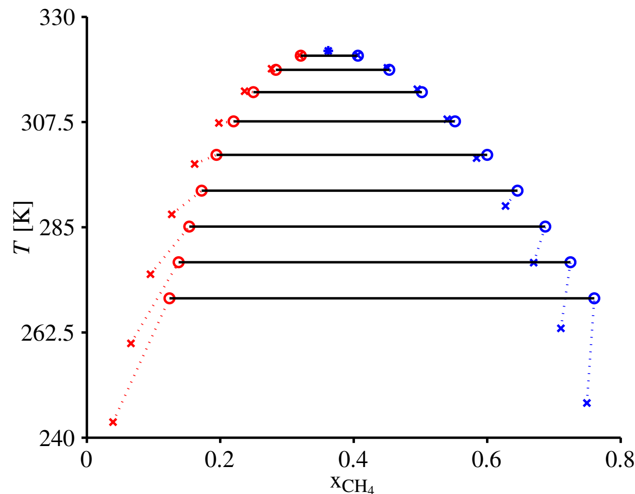


Figure 4.7: Prediction of mole fraction and temperature from the critical point of the system CH_4–H_2S. The critical point is indicated with an asterisk, the predicted points with a cross and the converged equilibrium points with circles. The predictions and the converged points are connected with a dotted line.

As can be seen from the Figure 4.7, the prediction is quite good close to the critical point as expected, but even far away from the critical point the prediction is good enough to be used as a start estimate for the correct solution.

4.4 Conclusion

The conditions for the critical and tri-critical points formulated in the unnamed energy potential $X(S,V,\mu)$ have been developed. The advantage with these criteria, compared to criteria in other energy potentials, is that they are independent of the number of components in the system. The SRK equation of state was used for calculating the results, and in order to make practical use of the energy potential X the Legendre transform from T, V,$\mathbf{n}$ to S, v, μ must be applied. Newton iterations are used for all the calculations, and since the solutions are already published it is possible to select the starting values such that the iterations converge easily.

Critical lines for the three binary systems CH_4–H_2S, CH_4–nC_6 and nC_6–H_2S were calculated with two different, albeit equivalent, critical point criteria. No difference in the stability or the computation time between the two approaches was experienced. This is as expected, since these criteria are thermodynamically equivalent.

The reported tri-critical point, of the system CH_4–nC_6–H_2S was recalculated, see Kohse and Heidemann (1993), and the convergence region of the Newton method has been reported. This shows that it in practice is not sufficient to use only the tri-critical criteria and a random starting value to locate the tri-critical point. The procedure shown here can only be used for the final steps. Other algorithms are needed to locate possible regions for the tri-critical point.

Furthermore, a Taylor expansion at the critical point of a two-component mixture was used to predict the phase boundary. This proved to work quite well, and it was indeed possible to predict a significant part of the phase boundary far away from the critical point. Temperatures 10% away from the critical point were calculated quite accurately, and served as good start estimates for phase equilibrium calculations.

The main purpose of this chapter was to verify that the automatic gradient and the automatic Legendre transform library from Part II produces correct results. From the figures shown in this chapter, and from comparisons with published results, it is concluded that the calculations are correct. It is taken for granted that it would be virtually impossible to locate a tri-critical point with erroneous derivatives. This is of course not a mathematically valid proof, but a proof of this kind is discussed in Chapter 6. From a practical point of view it is very important to know that the model is correctly implemented and that all the gradients up to the sixth order work smoothly in Newton-type iterations.

Part II

The RGrad Language

Chapter 5

Background

This chapter discusses some of the technologies that inspired the development of the software tool described in Chapters 6–8. Automatic differentiation, which is a tool for symbolic differentiation of computer code is presented, and in addition some of the possibilities for portable thermodynamics from commercial vendors are discussed.

5.1 Automatic Differentiation

Automatic differentiation (AD) is a technique aimed at differentiating computer code. The numerical results from AD is exact in the sense that the results are deviating only because of round-off errors. It is not a numerical method like divided differences, and is therefore free of truncation error. Even though the result from AD is as accurate as symbolic algebra, the computation is very different. While a computer algebra system (CAS) gives an expression that is intuitive to the human brain, AD focus on producing fast computing code. AD is also capable of differentiating computer algorithms with control flow statements (for, if, while, etc.), with some potential pitfalls. For instance, as pointed out by Kedem (1980) and Griewank (1989), the following expression poses a serious problem:

$$if \quad x \neq 0 \quad then \quad y = (1 - \cos x)/x \quad else \quad y = 0$$

If this expression is differentiated, and evaluated at 0, the numerical result is 0, whilst the correct answer is 1/2. There is no obvious solution to this flaw, other than restricting all expressions to be continuous and differentiable.

AD is based on the fact that, whatever the complexity of the function, it can always be broken down into a finite set of elementary operations (+, -, *, / etc.) and functions (sin, cos, exp, etc.). For instance, the simple function

$$f(x_1,x_2) = x_1 + x_1 * x_2$$

can be broken down to the elementary operations

$$\begin{aligned} v_{-1} &= x1 \\ v_0 &= x2 \\ v_1 &= v_{-1} * v_0 \\ v_2 &= v_{-1} + v_1 \end{aligned} \tag{5.1}$$

In the equations above the variables have negative indices and the expressions have positive indices. There are two fundamentally different programming approaches to differentiate program code, namely operator overloading and source transformations. Operator overloading means that the language parser is redefined such that the normal behaviour of the operators is changed. With operator overloading the derivative can be calculated runtime, at the same time as the original function is calculated. AD in the sense of operator overloading was first implemented by Kedem (1980), even though this was a hybrid approach using a Fortran precompiler. Operator overloading requires less implementation effort than source transformations, since it uses the parser as it is provided by the compiler. Operator overloading is therefore quite easy to implement and can can be applied runtime.

Source transformation produces new code, which must first be stored and then compiled. It is possible to use operator overloading for this approach as well, but in most cases the source is parsed by a program made for the purpose. The advantage is that the produced code will be significantly faster than the runtime calculations. In order to use operator overloading the host language must allow for this, while source transformation can be implemented in a language different from what the code is written in. For an extensive list of available tools, see Juedes (1991). Some of the well known AD tools are ADIFOR (Bischof et al., 1992a), ADOL-C (Griewank et al., 1996) and TAPENADE (Hascoët and Pascual, 2004). These tools are used extensively in for instance computational fluid dynamics and climate modelling (Bischof et al., 1992b; Kim et al., 2006; P. Heimbach and C. Hill and R. Giering, 2005)

The research of computer differentiation of computer code started in the Fifties, see for instance, Nolan (1953), Hanson et al. (1962), and the programming language LISP by McCarthy (1960). These early attempts were aimed at differentiating expressions in general, and are not directly comparable with modern automatic differentiation. Automatic differentiation can be divided into two different categories;

forward mode, and reverse (adjoint) mode. Forward mode was first introduced by Wengert (1964), and further developed by Rall (1981). Reverse mode dates back to Linnainmaa (1976) and Speelpenning (1980). The two modes will be discussed closer in the following sections.

5.1.1 Forward mode

In forward mode the derivatives are calculated in the same order as the original function is evaluated. If we look at Equation (5.1), forward mode AD gives:

$$\begin{aligned}
\dot{v}_{-1} &= \dot{x}_1 \\
\dot{v}_0 &= \dot{x}_2 \\
\dot{v}_1 &= \dot{v}_{-1} * v_0 + v_{-1} * \dot{v}_0 \\
\dot{v}_2 &= \dot{v}_{-1} + \dot{v}_1
\end{aligned}$$

Here, the dot means a derivative with respect to either x_1 or x_2. The value of x_n and its derivatives must be known beforehand in order to find a solution to the problem. Assuming that x_1 and x_2 represents scalars, and that the derivative with respect to any of them must be either zero or one, it is evident that $\partial y/\partial x_1$ can be represented as:

$$\begin{bmatrix} 1 & & & \\ 0 & 1 & & \\ -v_0 & -v_{-1} & 1 & \\ -1 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} \dot{v}_{-1} \\ \dot{v}_0 \\ \dot{v}_1 \\ \dot{v}_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.2}$$

Equation (5.2) is just a symbolic representation of how AD works. For practical details, look at Griewank (2000) and Mischler et al. (1995). The scheme is not implemented using matrix algebra because the solution of the system of equations is straightforward. Gilbert et al. (1991) gives two* different algorithms for performing forward automatic differentiation, with two fundamentally different ways of calculating the derivatives. In the first algorithm, the function and the derivative is calculated in parallel, while in the second algorithm a computer graph is generated first and later differentiated in a second sweep. The computer graph of the second algorithm requires more memory, but on the other hand higher order derivatives are possible. In order to calculate the gradient, the first algorithm needs to be called as many times as there are independent variables, while for the second algorithm, the computer graph can be reused, but it must be differentiated as many times as

*three algorithms are given, but the second is just an extension of the first one from calculating a single partial derivative to calculating the gradient, without recalculating the function value each time.

there are independent variables. See also Griewank (1989) and Griewank (2000) for a thorough discussion of this topic and other,alternative, algorithms. In their publications, specialized implementations considering the underlying properties of the program are considered to achieve highly optimized code. The computation of sparse Jacobians means for instance that zeroes in the equation system are handled symbolically and not as numeric zeroes.

Advantages

- As accurate as symbolic differentiation.
- Relatively easy to understand and to implement.
- Higher order derivatives can be calculated by applying the technique several times.
- Low overhead for single derivatives.
- Can be applied runtime.

Disadvantages

- Slow compared to reversed mode at calculating gradients for functions with a large number of variables.

5.1.2 Reverse mode

Reverse mode is obtained from Equations in (5.2) by transposing the matrix and changing the right hand side. The transposed matrix means that the order of the computer graph is reversed, hence the name reverse mode. The lower triangular Equations (5.3) are easily solved by back substitution. This is again not how this problem is solved in practice, it is just a convenient representation.

$$\begin{bmatrix} 1 & 0 & -v_0 & -1 \\ & 1 & -v_{-1} & 0 \\ & & 1 & -1 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \bar{v}_{-1} \\ \bar{v}_0 \\ \bar{v}_1 \\ \bar{v}_2 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{5.3}$$

The symbol $\bar{v}_n$ represents the adjoint value of v_n. v_0 and v_{-1} represents now the derivative of the function with respect to x_1 and x_2 respectively:

$$\begin{aligned}\bar{v}_{-1} &= v_0\bar{v}_1 + \bar{v}_2 \\ \bar{v}_0 &= v_{-1}\bar{v}_1 \\ \bar{v}_1 &= \bar{v}_2 \\ \bar{v}_2 &= 1\end{aligned}$$

Reverse mode of AD is performed in two operations; one forward sweep and one reverse sweep. The forward sweep works pretty much like forward mode AD when the computer graph is constructed. To find the gradient, the computer graph is traversed in reverse order, and the gradient* is obtained as the final solution of the function. Reverse mode is a lot less intuitive than forward mode. However, the cost of calculating the gradient is independent of the number of independent variables, which makes the method attractive for functions with many independent variables.

Advantages

- As accurate as symbolic differentiation
- Calculation of the gradient is independent of the number of independent variables, making this method very effective for systems with many variables.
- Can be applied runtime

Disadvantages

- Less intuitive than forward mode, making it more difficult to implement.
- Relatively large overhead for single derivatives.
- Require more memory than forward mode.

It looks like magic that reverse mode of AD achieves the gradient of the function within a constant factor of the evaluation time of the original function. There is, however, a simple explanation for this. First of all, most functions have a multiple input, single output characteristic. Reverse mode is calculating the sensitivity of the inputs with respect to the outputs. This happens to be the gradient, see

*The partial derivative of the function with respect to all independent variables

Griewank (2000). This means that it is sufficient to evaluate the function, apply the chain rule for each elemental expression, and back-calculate the adjoint function. If the number of outputs is of the same order of magnitude as the number of inputs, there is nothing to gain by using the reverse mode. For instance, if the gradient of a function is already calculated and the Hessian is wanted, the number of inputs and outputs are the same. In this case nothing is gained by applying reverse mode. Since the forward sweep of reverse mode is equal to the forward mode it is probable that forward mode is the method of choice in this case because there is no overhead in storing the intermediate results.

Finally, both forward and reverse mode of automatic differentiation delivers exact results, which is a big advantage over numerical differentiation procedures.

5.1.3 Linear algebra expressions

The previous description is only valid for scalar operations, but it would be desirable to apply automatic differentiation to linear algebraic expressions as well. At first, this seems like a straightforward task by applying the same techniques as for the scalar operations. The problem is that scalar operations are commutative, while linear algebra expressions are not[*].

If the linear algebraic expression is differentiated with respect to a scalar variable, it is always possible to construct the derivative. The challenge is to recognize the scalar in the linear algebra context (imagine the scalar as one element in a vector or a matrix). Such an approach should be possible, but the author is not aware of any applications. It can then be concluded that this is probably not worth the effort, probably because it requires a tremendous amount of bookkeeping.

Another approach, which has been reported by Kalaba et al. (1987), is to break the problem down into binary operations, and use standard automatic differentiation to differentiate these expressions. The result is afterwards put back into vectors and matrices. In this way they manage to keep the structure of the original problem. This is done by modifying the existing automatic differentiation routine to handle matrix operations. These FORTRAN subroutines actually break all expressions down to scalar operations. The approach is limited by the size of the result, however, and it cannot be used for more than two-dimensional arrays.

[*]commutative means that A@B = B@A, where @ is an operator. For instance in linear algebra: $A*B \neq B*A$.

5.2 Portable Thermodynamics Software

This section summarizes portable thermodynamics from a few commercial companies. Due to time constraints and license issues none of the programs have been tested. This is therefore not a survey of features available in the different thermodynamic packages, but rather a survey of what possibilities are available, and what technology is used to make thermodynamic calculations available outside the environment it was created for. The software mentioned here is a mere selection of available possibilities. It does, however, cover the most common ways to supply portable thermodynamics in different user environments.

5.2.1 Computer-Aided Process Engineering

CAPE-OPEN (CO) is not a software tool from any company, but a set of standards which enables program interoperability between software components. The CO standards is not limited to thermodynamics, but helps the user in linking different software component for use in modelling and process simulation. The standards are maintained by a non-profit organization called CAPE-OPEN Laboratories Network. The thermodynamic and physical properties interface specification (CO-LaN, 2006) defines how the most common thermodynamic quantities are communicated between different software components. The standard defines the names of the different elements that are included. This set of names is closed within each version of the standard, but can be subject to changes from one version to the next. The same function has then the same interpretation in all CO applications. This means that it is possible to switch from one CO compliant application to another without changing the user program. The standards define the interfaces, not the internal structure, which can be whatever the programmer desires. It can therefore be realized in almost any language. The CO components are connected through a middle software layer which handles the communication. Currently two different alternatives are available, COM* for the Windows platform, and CORBA on UNIX/Linux platforms.

5.2.2 Thermo-Calc

Thermoc-Calc is a software package for thermodynamic calculations originating from KTH in Sweden. It is especially adopted for calculating phase diagrams and connected problems in material science.

*Component Object Model

Thermo-Calc supplies two alternative programming interfaces. The TQ interface is written in FORTRAN and is for FORTRAN applications. This can be used on Windows, Linux and UNIX platforms. The code will make use of the Thermo-Calc engine for calculations. This requires that the Thermo-Calc software has been installed.

The other interface, TC-API, is a C interface to the Thermo-Calc engine. This interface has more features available than the TQ interface, but does not seem to have UNIX support. The interface has been used by the company to provide both a Windows graphical user interface and a Matlab interface to the Thermo-Calc engine.

5.2.3 Simulis thermodynamics

Simulis thermodynamics is a software tool, from the company Prosim, that enables thermodynamic property calculations for both single component and multicomponent phase equilibrium calculations.

The tool is available on Microsoft Windows platforms that support the Microsoft COM/DCOM* technology. This solution includes a vast number of programs and programming languages, like for instance Microsoft EXCEL, Matlab, C++, FORTRAN etc. A thermodynamic system with all the components and a proper model description must be prepared through the graphical user interface before the calculations are done. Simulis thermodynamics is also CAPE-OPEN compliant. This means that it is possible to generate CO property packages for use in other CO compliant software, or other CO compliant thermodynamic software can be used inside Simulis thermodynamics. However, all calculations require that the Simulis thermodynamics software is installed on the computer.

5.2.4 Aspen Technology

Aspen Technology provides a wide range of different portable thermodynamics alternatives. Both Aspen Plus and Aspen HYSYS simulators are CAPE-OPEN compliant, which enables the user to apply the thermodynamic engines in other CO compliant software. In addition two other interoperability softwares are provided.

COMThermo is a thermo-physical property calculation engine which utilises Microsoft's COM technology for interoperability. This enables other Microsoft Win-

*Distributed Component Object Model

dows programs enabling COM plug-ins to use this package. COMThermo is also CO compliant.

The last alternative from Aspen Technology is Aspen Properties. In addition to COM and CO capabilities, a FORTRAN interface is supplied.

5.2.5 Multiflash

Multiflash is a multiphase equilibrium package, and is supplied as a dynamic linked library for linking with other Windows programs. It is possible to include Multiflash in Visual Basic, Java, C/C++ and FORTRAN programs. In addition, a CAPE-OPEN interface is available, and an interface to Matlab too. It should be noted that Multiflash has been embedded into the modelling system gPROMS.

5.3 Remarks

Automatic differentiation looks very promising, but since our project aims at generating gradients to infinite order with respect to multidimensional input arrays, it was decided to follow a different strategy. It has been concluded that automatic differentiation require too much bookkeeping to be suitable for the task, and in the following chapters an alternative to automatic differentiation is developed. The differentiation scheme which is developed is capable of calculating gradients to infinite order without the bookkeeping required for the approaches presented in the current chapter.

The variety of methods for providing portable thermodynamic calculations that are presented in this chapter inspired the portable thermodynamics tool described in Chapters 6–8. The methodology presented in this thesis is fundamentally different from any of the commercial programs because the commercial vendors require the installation of both a calculation engine and a database before the thermodynamics calculations are available to the external program, whereas the present approach is a stand-alone solution. This means that a thermodynamics package must be installed and there must be a valid license for the portable thermodynamics to work. The tool presented in this thesis aims at generating a standalone program (library) for each individual problem. The great advantage is that this code has no dependencies. As a first approach to creating portable thermodynamics it was decided to generate C-code with a generic C-API. This opens for the possibility of implementing various kinds of of portable thermodynamics interfaces at a later stage, adapted to several different languages.

Chapter 6

Language Definition

The RGrad language presented in Chapters 6-8 is the result of a programming adventure where neither the goals nor the technical solutions were given in advance, and where some of the choices had to be made without knowing the exact consequences they would have for the development. The material in this section illuminates the important crossroads in the development of the code.

The most important choice in the development, which also distinguishes the RGrad language from other similar methodologies, is the differentiation technology. Instead of using well known techniques which produces scalar derivatives, a novel method for calculating symbolic gradients was developed. The method proved to be a big success, as it enables higher-order gradients in a very simple manner. It must be stressed that the symbolic gradient calculation is different from, and more complicated than, the scalar differentiation that is routine in CAS[*] and AD[†]. In fact, the achievement of symbolic gradients more than compensates for reinventing the wheel. First of all the algebra must enable commutative multidimensional array operations. This is possible through a technique called broadcasting, which is used to conform a set of non-conforming arrays, as will be explained later in this chapter.

The specific choice of programming language is of course important for the layout of the program code, but it is not important for the fundamental concepts developed in the thesis. Similar results would indeed be possible in most modern programming languages, but the choice of using the high-level interpreted language Ruby is nevertheless important for the realization of the project. The exploitation of a

[*] Computer Algebra Systems
[†] Automatic Differentiation

tailor-made modelling language combined with a flexible use of the Ruby parser saved a lot of implementation efforts without having any negative side effects.

ANSI C was chosen as the language for the final product. This is immaterial for demonstrating the concepts of automatic gradient calculations and for the code generation, but it is important for the versatility of the final product. ANSI C-code can easily be integrated with other programs with a minimal amount of effort. Of course other languages like FORTRAN might be possible, but it requires in general more programming effort because C has become the de facto standard in low-level platform independent programming.

6.1 Introduction

In thermodynamic calculations both the gradient and the Hessian of the energy potential are used extensively. Equilibrium flash calculations is a typical example of the use of a thermodynamic potential surface. In order to solve the flash, the intensive properties of the phases must be equal, that is, the temperature, the pressure and the chemical potentials of each component must be equal in all the phases. However, most common thermodynamic models are implicit in pressure and chemical potentials. Actually, thermodynamic models can be divided in two main groups: Equations of state, with natural variables temperature, volume and mole numbers; and activity models with natural variables temperature, pressure and mole numbers. The chemical potentials are derived by differentiation of the energy potential, but because some of the quantities are implicit the final equilibrium problem must be solved iteratively.

Most equation solvers require access to the Jacobian to solve the equilibrium problem, but since some of the implicit variables are first-order derivatives of the energy potential the Jacobian will simplify to the second-order gradients (Hessian). Most solvers will calculate the Jacobian numerically, or may also take advantage of a pre-computed Jacobian. The problem with numerical differentiation is that the accuracy decreases significantly compared to symbolic expressions. The great advantage, on the other hand, is that the technique is very easy to implement. Traditionally, the Jacobian is derived by hand-coding of the derivatives, but this is time consuming and error prone. The advantage is that it is possible to make clever simplifications by exploiting problem specific properties, and thereby produce highly optimized code. Both approaches have limiting disadvantages, and for this reason alternative approaches were sought.

In seeking a new approach for calculating the derivatives, certain properties should be fulfilled by the method:

- As accurate as hand-coding.
- Algorithmic approach, i.e. no human interaction.
- Possible to differentiate to infinite order.
- Possible to differentiate with respect to both variables and parameters.
- Keep the structure defined by the problem.

At first automatic differentiation (AD) was considered (Griewank, 1989; Juedes, 1991), see Chapter 5 for more details. This method can, either by runtime methods like operator overloading or by source code transformations, create an exact derivative based on the source code. However, the main focus in the present work has been on the differentiation of thermodynamic models with a high degree of structure[*]. AD is based on differentiation of scalar operations, which means the structure must be dealt with in order to differentiate it. Once the structure is removed, it is very difficult to reconstruct. The task is therefore to find a method which makes it is possible to differentiate algorithmically, and at the same time keep the structure of the model.

When a structured model is differentiated with respect to a variable set, the structure of the gradient is a combination of the structures of the model and the variable. The structures are combined and the number of dimensions is thereby increased, and to keep this structure is a major challenge. A simple example is a vector of mole numbers which is differentiated with respect to itself. This should give an identity matrix. If further differentiation were possible, the second-order gradient should be a three dimensional zero matrix and so forth for the fourth and higher-order gradients.

Another aspect of differentiating thermodynamic models is that the variables are usually divided into three disjoint sets, where two of the sets contain single scalar variables, and the last set consists of one or more variables[†]. It must be possible to calculate the gradient of the model with respect to any of these sets, regardless of the size. It must also be possible to calculate the gradient to arbitrary order, and with respect to any combination of the variable sets.

[*]By structure it is meant an element which holds more than one value. For instance, a scalar value will be an element without structure, while a vector or a matrix will have structure.

[†]In a thermodynamic model these sets typically represents temperature, volume or pressure, and mole numbers.

The structure of most thermodynamic models can easily be captured by using linear algebra. A typical model includes terms like $f = x^{\mathrm{T}}Ax$, where f is a scalar value, x is a vector, and A is a matrix. In most cases it is possible to write both the gradient and the Hessian of these expressions as linear algebra expressions. This, however, requires human interaction and prior knowledge to the problem. It is difficult, if not impossible, to create an algorithm for doing these operations in a general manner. Hand-coding of the derivatives is by the way very time consuming and requires a lot of experience in linear algebra, and a firm knowledge of thermodynamics (especially for verification and debugging).

It is difficult to calculate the gradient of arbitrary linear algebraic expressions and at the same time retain something that makes sense in a linear algebra context. In order to use automatic differentiation, the expressions must be broken down to a series of binary operations, which are differentiated using standard techniques. It is then very difficult to get back a structured expression, and since it seems impossible to differentiate the linear algebra expressions directly, a different approach for capturing, and maintaining, the structure is required.

As mentioned earlier, even simple thermodynamic models requires matrix multiplication. This multiplication has certain features which makes it very useful, but the same features are what make differentiation so difficult. In matrix multiplication each element of the rows of the left operand are combined with the corresponding elements of the columns of the right operand. Afterwards the products are summed and copied into the resulting matrix. This means that two different algebraic operations are done simultaneously. Another issue about matrix multiplication is that it is non commutative, that is, $AB \neq BA$.

A straightforward way of looking at the matrix product AB is to combine each row of matrix A with each column of matrix B, then calculate the element by element product of the row/column and sum the result into a scalar value. This procedure creates temporarily a three-dimensional structure, which is reflected in the $O(n^3)$ runtime for a naive implementation of the matrix multiplication. The problem can be split into two separate procedures, one procedure for combining the right elements, and one procedure for defining the scalar operations. As will be explained later, this makes it possible to differentiate the structured expression.

The generalised matrix operation can be implemented as an iterator* keeping the structure of the problem, and an operator working on the scalar values. The matrix multiplication depends on both, but there is no direct link between the iterator and the operator. This means that the iterator can be differentiated independent

*Iteration is here meant as an algorithm for traversing a structure in a systematic way. It has nothing to do with numerical iteration

of the operator. The differentiation of the scalar operator is a relatively simple task, but the differentiation of the iterator is not so trivial. In fact, defining the iterators used for matrix multiplication is maybe even not so straightforward. The iterator must be sufficiently general to capture all possibilities needed to describe a thermodynamic model, and its associated derivatives.

In the search for a general iterator it is natural to look at languages which have their strong-points in the handling of multidimensional arrays; like APL, (Falkoff and Iverson, 1973; Willhoft, 1991; Iverson, 1991; IBM, 1994), Yorick and Python (with Numerical Python) (Ascher et al., 2001).

APL defines inner and outer product iterators which lets the user decide the operator(s) to be used. In addition, a rank reduction operator is defined, which reduces the rank of the array. The outer product takes each item in turn from the two operands, and combines them element by element, regardless of the rank of the operands. The operator to be used when combining the two items must be provided separately. As an example; if the operator "$+$" is given then the two scalars are added, if the operator "$*$" is given the two scalars are multiplied, and so on. The "reduce" function lets the user decide which operator should be used in the reduction, and it is also possible to define which dimension that should be removed. The inner product iterator makes combined use of the outer product iterator and the "reduce" iterator. The iterator will combine the sub-arrays along the inner most dimension of the left operand with the sub-arrays along the outer most dimension of the right operand. This is done with the outer product iterator, and the "reduce" iterator is used on each item of that result. The operators used in the outer product and the rank reduction must be given by the user.

Yorick achieves multidimensional inner and outer products by clever indexing. The indexing makes the iterator a bit more flexible compared to APL, but it is not possible to define the operators for the inner product. Something called pseudo-indexes can be used to extend the array dimensions without copying the elements, or even without creating a new array. What this pseudo-index does, is to add a dimension of length one in the direction indicated. When operating on two arrays the dimensions are read from left to right, and as long as the dimensions of the two arrays are equal the usual operation is performed. If the length of the dimension of either array is unity, or missing, the elements of that array are repeated to conform the two arrays. This technique is called broadcasting. The pseudo-indexes make it possible to write commutative outer products, but the inner products are more restrictive than the ones in APL. While APL allows the operators to be defined by the user, the inner product only works with multiplication in Yorick. When the multiplication operates on two arrays a pseudo-index can be used to tell which of the dimensions that should be reduced. One pseudo-index must be given for each

array, and the length of the dimension to be reduced must be equal for both of the arrays. It is not possible to reduce more than one dimension at a time.

Numerical Python works much in the same way as Yorick, but like APL it has a more general rank reduction function. With this function it is possible to define the binary operation to be used in the rank reduction. The reduce function will only reduce the array by one dimension at a time, but it is possible to tell which of the dimensions that should be removed.

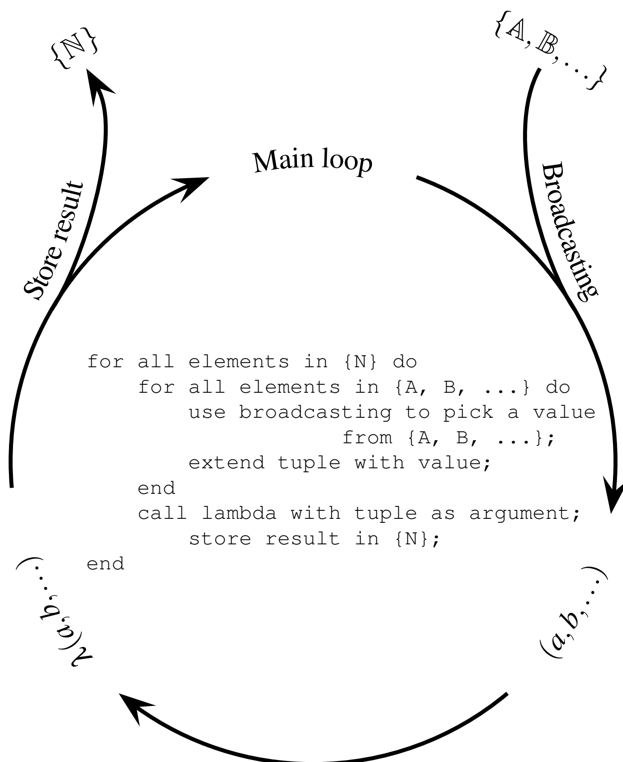


Figure 6.1: A list of set containers $\{\mathbb{A}, \mathbb{B}, \ldots\}$ are given to the iterator which picks one element from each and creates an n-tuple. This is given to the λ-function, and the result is stored in a new set container $\{\mathbb{N}\}$.

By combining these methods, and generalising the concept further, it is possible to define an iterator which iterates on a finite, but possibly large set of arrays, and applying the scalar values from the iteration on a function also defined outside the iterator. This is exemplified in Figure 6.1. A further discussion of the procedure is given in Section 6.2. What makes this approach different from the others mentioned is that there are no restrictions on the number of arrays operated upon by the

iterator. In addition the operations between the elements of the arrays are given as a separate λ-function. The only restriction is that this function must take the same number of arguments as there are arrays in the iterator. The iterator will then pick one element from each array and feed them to the λ-function. There are no restrictions on the rank reduction, and the dimensions that are removed are simply given to the rank reduction method. This can be applied both in combination with the general iterator, and as a separate function call.

6.2 Grammar Definition

In Section 6.1 it was suggested to build a new broadcasting concept based on the ideas of languages such as APL, Yorick and Numerical Python to create a general iteration procedure which satisfies the requirements listed in the beginning of this chapter. It was decided to formalise these ideas into a small grammar. The Extended Backus-Naur form (EBNF) (ISO/IEC 14977:1996, E) grammar is shown in Figure 6.2.

The grammar first defines a Model, where a Model can consist of one or many Expressions. A graph representation of a simple model is shown in Figure 6.3. As seen from the figure, all expressions defined directly in the Model are root nodes in their own subtree. Each Expression consists of other Expressions, Variables and Parameters. The Variables and Parameters are represented as leaf nodes in Figure 6.3, while Expressions are subtrees. Each Expression consists of three mandatory members, and one optional member, namely a set of elements, a λ-function, and a set of dimensions. In addition an optional sum can be defined. The set of elements consists of one or more Variable, Parameter or Expression. Each element has the broadcasting mechanism attached, where the broadcasting belongs to the elements, but is set in the context of the Expression. The broadcasting consists of a set of zero, one or many true and false statements. A thorough description of broadcasting is given in Section 6.4.

Within the grammar, there is no difference between Parameters and Variables, but to be able to distinguish them later, they are defined separately. Each Parameter or Variable consist of a set of numbers, a set of dimensions, and an optional sum. The set of numbers must contain at least one member, but there can be several members. The set of dimensions can have zero, one or many members. If the set of dimensions has zero members, it is required for the set of numbers to hold only one number. This represents a scalar value within the grammar[*].

[*] Scalars, vectors and matrices are defined respectively, with the size of the dimensions in paren-

$$
\begin{aligned}
Model =& Expression\{`,` Expression\};\\
Expression =& VPE\ \ lambda\ `,`\ Dim[\ `,`\ Sum];\\
VPE =& (Variable\ `,`\ Bc\ `,`)|(Parameter\ `,`\ Bc\ `,`)|\\
& (Expression\ `,`\ Bc\ `,`)|\{VPE\};\\
Bc =& `\emptyset`|bool|\{bool\};\\
lambda =& Unary|Binary|Ternary|n;\\
Unary =& uop\ lambda;\\
Binary =& lambda\ bop\ lambda;\\
Ternary =& `if`(lambda\ >\ lambda)\ `then`\ lambda\ `else`\ lambda;\\
Variable =& Numbers\ `,`\ Dim[\ `,`\ Sum];\\
Parameter =& Numbers\ `,`\ Dim[\ `,`\ Sum];\\
Sum =& Bc, sum;\\
sum =& n\{\ `+`\ sum\};\\
Numbers =& n\{\ `,`\ n\};\\
Dim =& `\emptyset`|Digits\{\ `,`\ Digits\};\\
n =& Digits\ `.`\ Digits;\\
Digits =& digit\{digit\};\\
digit =& 0|1|2|3|4|5|6|7|8|9;\\
bool =& `true`|`false`;\\
uop =& `+`|`-`|`\cos`|`\cos^{-1}`|`\cosh`|`\cosh^{-1}`|`\exp`|`\ln`|`\log`|\\
& `\sin`|`\sin^{-1}`|`\sinh`|`\sinh^{-1}`|`\mathrm{sqrt}`|`\tan`|`\tan^{-1}`|`\tanh`|`\tanh^{-1}`;\\
bop =& `+`|`*`|`-`|`/`|`**`|`\%`;
\end{aligned}
$$

Figure 6.2: EBNF grammar of the RGrad language.

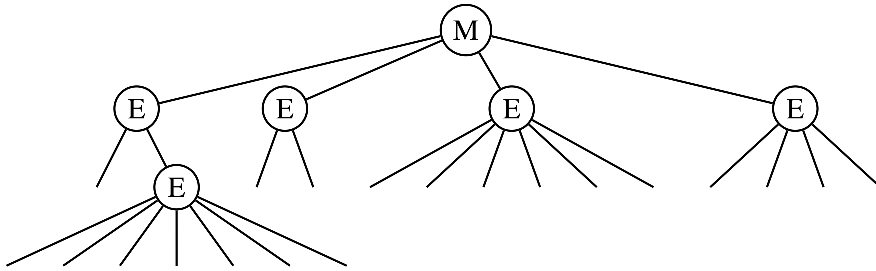


Figure 6.3: Example of how a Model graph can be built up. Empty nodes represents leaf nodes. M means Model, and E Expression.

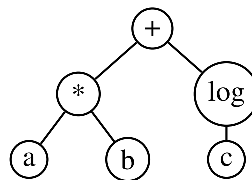


Figure 6.4: Tree representation of a λ-function. Should be read left to right with infix operator.

The λ-function is loosely related to the λ-calculus developed in mathematical logics in the 1930s by Church (1932, 1941) and Kleene (1935). It has later become the basis of functional languages, which is the reason why the name λ-function is used for the anonymous function needed in the current context. The λ-function is nothing more than a computation tree which can be either a unary tree, a binary tree, a ternary tree or a pure number. A unary tree consists of a unary operator and a tree. The unary operator can be any unary scalar operator defined. The binary tree consists of one binary operator and two nodes, which can be other trees, or leaf nodes i.e. numbers. The binary tree is written in left-to-right infix notation. The binary operator can be any scalar binary operator. As an example, the tree representing the function body $a*b+log(c)$ is shown in Figure 6.4.

Only one ternary operator is implemented, namely the if-else test. This test compares two trees, and the true statement is returned if the first tree is larger than the last tree, otherwise the false statement is returned. There is a one-to-one correspondence between the elements of an expression and the elements of the λ-function, with the exception of program constants included directly in the λ-function. This is explained in more detail in Section 6.4.

The sum consists of two parts, one broadcasting part and one operator part. The

thesis. $s=1$ $(\emptyset), s=[1,1]$ $(2), s=[[1,1],[2,2]]$ (2×2)

broadcasting in the sum is used to tell which of the dimensions should be summed. The operator is fixed, and implements the "+" operator, which will give the sum of the elements.

This small grammar is sufficient to capture the structure of most thermodynamic models. In order to satisfy all the requirements listed on Page 77 the grammar must be closed. Arguments for this is given in the next section. However, thermodynamic models with implicit terms cannot be represented with this grammar.

6.3 Gradient Calculations

An important design criterion for the grammar shown in Figure 6.2 is that the gradient should be expressible within the same grammar definition as the parent function. In order to differentiate a Model, all parts must be differentiable, and the result must be within the grammar. A Model can be differentiated with respect to both Variables and Parameters. Both Variables and Parameters can represent multiple values, and differentiating with respect to the entire structure yields the gradient. The gradient of the Model extends the model with more Expressions, and if the gradient of the Model does not exist the Model remains unchanged. In order to prove that the gradient of a Model is within the grammar definition, all gradients of the Model must also be within the grammar definition. It is important to note that the gradient calculation itself is not a part of the grammar, but the results from the gradient calculations must still be conceivable within the grammar.

The chain rule is applied whenever an Expression is differentiated. As mentioned in Section 6.2, there is a one-to-one correspondence between the elements of the Expression and the variables in the λ-function. To satisfy the chain rule the λ-function must be differentiated with respect to all the arguments. Each derivative results in a new Expression, which inherits the set of elements from the original Expression. However, the one-to-one correspondence between the elements in the set and the variables in the λ-function must still be satisfied. If the differentiation of the λ-function reduces the number of arguments, the argument set must be reduced accordingly. A mathematical representation of this problem is shown in Figure 6.5.

When the λ-function is differentiated there are three possible outcomes. In addition to the reduction already mentioned, the λ-function can be entirely removed. This means that the gradient does not exist, and no new Expression should be created. Since the λ-function is always differentiated with respect to one of its arguments this should never happen but it is required for completeness. The last

$$\varepsilon : \mathbb{B}, \mathbb{B}, \ldots \overset{\lambda}{\curvearrowright} \mathbb{B}$$

all $\mathbb{B}$ have same dimension(s)

Run-time use of $\lambda : \mathbb{Q}^{(1)} \times \mathbb{Q}^{(1)} \times \ldots \curvearrowright \mathbb{Q}^{(1)}$

Chain rule (c.r.) applied;

$$\frac{d\lambda}{d\alpha} = \frac{\partial \lambda}{\partial x}\frac{dx}{d\alpha} + \frac{\partial \lambda}{\partial y}\frac{dy}{d\alpha}$$

$$\hat{=} \lambda_x \frac{dx}{d\alpha} + \lambda_y \frac{dy}{d\alpha}$$

λ_x and λ_y are used to create new expressions $\varepsilon_x : \mathbb{B}, \mathbb{B}, \ldots \overset{\lambda_x}{\curvearrowright} \mathbb{B}_x$

$$\varepsilon_y : \mathbb{B}, \mathbb{B}, \ldots \overset{\lambda_y}{\curvearrowright} \mathbb{B}_y$$

These expressions are used to create a new expression, where the chain rule is applied. $\mathbb{B}_{\nabla_\alpha x}$ and $\mathbb{B}_{\nabla_\alpha y}$ are assumend known.

$$\nabla_\alpha \varepsilon : \mathbb{B}_x, \mathbb{B}_{\nabla_\alpha x}, \mathbb{B}_y, \mathbb{B}_{\nabla_\alpha y}, \ldots \overset{\text{c.r}}{\curvearrowright} \mathbb{B}_{\nabla_\alpha}$$

Figure 6.5: Differentiation of an Expression explained in text.

possibility is that there are no variables left in the λ-function after differentiation, only numerical values. This situation must be handled within the grammar. Since there are no variables left in the λ-function, it is not possible to create a new Expression. There are at least two ways to solve this problem, both of them within the grammar definition: The first solution is to incorporate the scalar value into the λ-function which will be created as a result of the gradient calculations. The second solution is to create a Parameter, which has the same numerical value for all the elements needed to satisfy the dimensions of the original Expression.

To satisfy the chain rule, as shown in Figure 6.5, the newly created element must be multiplied with the corresponding gradient, and the computation resulted from the chain rule must be represented within the grammar definition. This is solved by creating a new Expression where the λ-function reflects the sum of products. This product is removed from the Expression if one of the terms of the chain rule is zero. If all the products are removed, the gradient of the Expression does not exist, and the new Expression is in this case not created.

Next, the gradients of the Variables and Parameters needs to be determined. Since Variables and Parameters have equal implementations, the deduction of the gradient will be the same. In fact, there is only one case where the gradient does exists, and this is when the gradient is calculated with respect to itself. The result

is then unity, and for all other cases the gradient is zero. When the Parameter or the Variable represents more than one scalar value, the gradient will be calculated with respect to all the elements in the structure. This gives a new structure with 1 along the diagonal, quadragonal and so forth, and 0 elsewhere. This can be represented by creating a new Parameter. This means that the gradient of a Variable or a Parameter can be represented within the grammar definition.

When a summed Expression, Variable or Parameter is differentiated nothing special happens to the sum. The sum is inherited by the created Expression, and the broadcasting is expanded. This operation is possible because the gradient of a sum is the sum of the differentiated elements.

In this chapter it has been explained that the gradient of a Model can be constructed within the grammar. Since this is true for any gradient, it will also be true for higher-order gradients, and the grammar is therefore said to be closed with respect to gradient calculations.

6.4 Broadcasting

A technique called broadcasting for handling structured data of higher-order (rank > 2) is mentioned in Section 6.1. There it was concluded that linear algebra complicates the differentiation, and it was suggested to use broadcasting instead. This section gives an overview of how broadcasting is implemented for the grammar described in Section 6.2.

Broadcasting is used only to combine elements of arrays in a regular manner. No numerical operations are done, and broadcasting must therefore be combined with other techniques. The suggestion is to send the tuples of combined elements to an anonymous λ-function which calculates the final result. The advantage is that the λ-function can do common operation like $+$, $*$, $-$, etc., in addition to more advanced function expressions like for instance $a + b * log(c)$.

The task is to combine arrays of different shapes[*] and sizes[†] in a systematic way. In this respect broadcasting is used only to conform the arrays. The arrays are then combined element-by-element and sent to a λ-function to get the desired result. Instead of creating a copy of each array with the correct shape and size, the elements that needs to be repeated to conform the array is reused.

[*] Shape refers to the structure of the arrays, reflected by dimensions

[†] Size refers to the total number of elements in the array, regardless of shape.

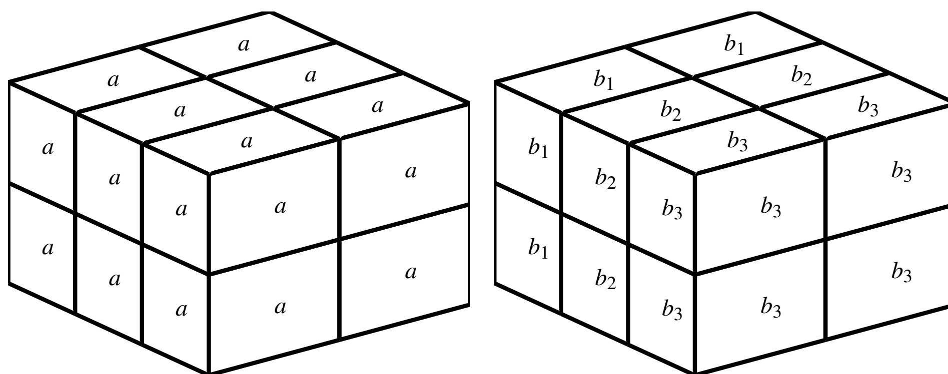
What makes broadcasting different from other methods of combining arrays of unequal shape, like for instance linear algebra, is that there are no restrictions on the shape, size or the number of arrays that can be handled at a time. The only restriction is that the arrays must conform after the broadcasting.

If an array has the same shape and size as the desired result, there is no need for broadcasting. In all other cases broadcasting must be specified by the programmer, and all arrays that shall be combined must have the same size and shape after the broadcasting is done. Only the programmer knows what the result should look like, and the broadcasting must of course be specified accordingly. Note that broadcasting only changes the dimensions of the array, not the numerical elements. The arrays are pealed off like an onion where the outer dimension corresponds to the first shell and the actual numerical elements are found in the center. For each array the programmer must start with the outer dimension and make sure that all remaining dimensions are equal. When an array is broadcasted there will be several combinations of broadcasting possible that yield different results. The model equations determine what dimensions should be broadcasted, and when a dimension is broadcasted it means that all the sub-arrays are repeated for the elements of the broadcasted dimension.

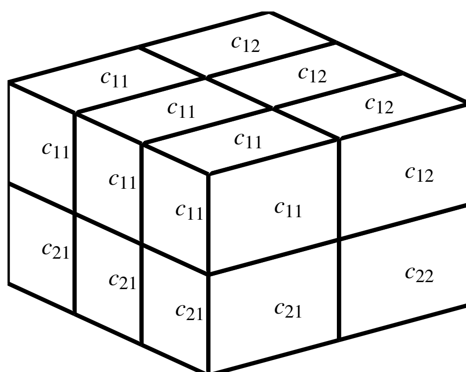
Figure 6.6 shows by example how broadcasting works. The broadcasting starts with the outer dimension and works its way inwards, one dimension at a time. Note that except for the last dimension all the elements in the array are sub-arrays. Each sub-array inherits the remaining dimensions of the parent array, including the broadcasting. The inheritance continues until all the dimensions are exhausted. For each element of the resulting array, one scalar value from each of the combined arrays is collected into an n-tuple. There will then be as many n-tuples as there are elements in the left hand side array. The n-tuples are sent to the λ-function and the result is stored in a new array.

This can be illustrated with a few short examples: Let the first be without broadcasting, where the 2×2 arrays $[[a_1,a_2],[a_3,a_4]]$ and $[[b_1,b_2],[b_3,b_4]]$ shall be combined. The outer dimension is handled first, and for each of the arrays a and b there are two sub-arrays, so now the arrays $[a_1,a_2]$, $[b_1,b_2]$ and $[a_3,a_4]$, $[b_3,b_4]$ have to be combined. Finally, the 2-tuples $[[(a_1,b_1),(a_2,b_2)],[(a_3,b_3),(a_4,b_4)]]$ are created and sent off to the λ-function, one pair at the time.

Let us redo the example with array b replaced by $[c_1,c_2]$. The two arrays a and c are now of different shape, and broadcasting is essential. In this case, there are two different ways of broadcasting the one-dimensional array, but the two alternatives give different results. If array c is broadcasted in the first dimension the whole array will be combined with each sub-array of array a. This gives $[a_1,a_2]$, $[c_1,c_2]$



(a) Scalar a broadcasted in all directions. (b) Vector b broadcasted in two directions.



(c) Matrix c broadcasted in one direction.

Figure 6.6: Graphical representation of broadcasting.

and $[a_3,a_4]$, $[c_1,c_2]$, which combines to $[[(a_1,c_1),[(a_2,c_2)]$ and $[(a_3,c_1),(a_4,c_2)]]$. The other option is to broadcast array c in the last dimension. After the first dimension is handled this gives $[a_1,a_2],c_1$ and $[a_3,a_4],c_2$. Next c_1 and c_2 are broadcasted to all the elements of the respective sub-arrays which combine to $[[(a_1,c_1),(a_2,c_1)],[(a_3,c_2),[(a_4,c_2)]]$. The two examples corresponds to Ac and $c^{\mathrm{T}}A$ in matrix algebra.

Finally, a more elaborate example is given, where the arrays $[a]$, $[b_1,b_2,b_3]$, $[[c_{11}, c_{21}],[c_{12},c_{22}]]$ are combined. The arrays a, b and c have all different sizes and shapes. In order to conform them, they must all be broadcasted. The final array has dimensions 2×3×2, which means array a must be broadcasted in all three dimensions, and array b must be broadcasted in the first and third dimensions, and

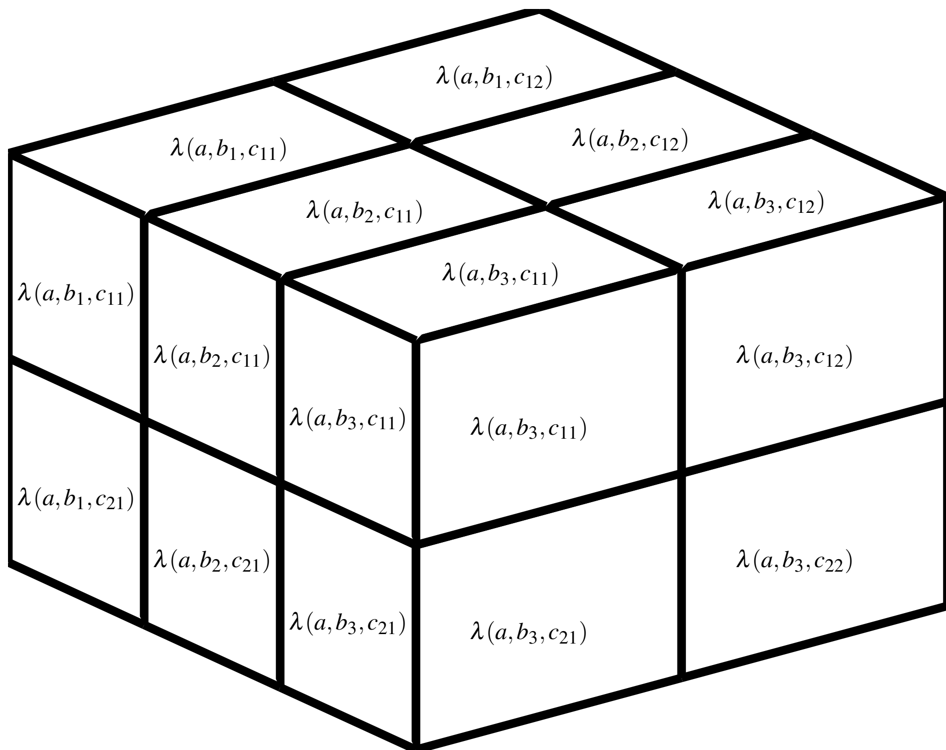


Figure 6.7: When the arrays of Figure 6.6 are combined, the same λ-function is used for all elements.

array c must be broadcasted in the second dimension.

After the first dimension has been resolved, a and b are broadcasted:

$$[a],[b_1,b_2,b_3],[c_{11},c_{21}],$$
$$[a],[b_1,b_2,b_3],[c_{12},c_{22}]$$

Next a and c are broadcasted:

$$[a],[b_1],[c_{11},c_{21}],$$
$$[a],[b_2],[c_{11},c_{21}],$$
$$[a],[b_3],[c_{11},c_{21}],$$
$$[a],[b_1],[c_{12},c_{22}],$$
$$[a],[b_2],[c_{12},c_{22}],$$
$$[a],[b_3],[c_{12},c_{22}]$$

Finally, the 3-tuples are combined:

$$
\begin{aligned}
&[[[(a,b_1,c_{11}),(a,b_1,c_{21})],\\
&\quad[(a,b_2,c_{11}),(a,b_2,c_{21})],\\
&\quad[(a,b_3,c_{11}),(a,b_3,c_{21})]],\\
&\;[[(a,b_1,c_{12}),(a,b_1,c_{22})],\\
&\quad[(a,b_2,c_{12}),(a,b_2,c_{22})],\\
&\quad[(a,b_3,c_{12}),(a,b_3,c_{22})]]]
\end{aligned}
$$

The series of operations is summarized in Figure 6.6 and Figure 6.7. As can be seen from Figure 6.7 the same λ-function is used for all the elements. This is maybe the strongest point of broadcasting. Rather than using a combined operation like AB for the matrix (inner) product, we can write something like $[A,B] \overset{\lambda}{\curvearrowright} C$ where the λ-function can take virtually any action on the pairs $c_{ij} = \lambda(a_{ij}, b_{ij})$.

Chapter 7

Model Generation

The grammar described in Chapter 6 defines a set of rules according to which the computer code must be written. The goal is to code a thermodynamic model, differentiate it, and produce highly flexible computer code. Flexible in this context means how it can be used, and where the code can be used. The differentiated thermodynamic model is held in a computer graph, which makes it relatively easy to create native code in most computer languages. Even though it is possible to generate native code in different languages, the main focus here has been on creating ANSI C-code. The C-code can be linked into most modern languages by using the native language's API, or directly into third party programs by using a run-time information exchange standard such as the COM interface.

The exportation can be described in three separate parts: Part one, discuss how the model should be represented such that the last two parts are possible. Part two defines an internal language which makes higher-order differentiation easy. Part three use the graph created and generate the output. The first two parts are described in this chapter, while the third part is described in Chapter 8.

Throughout this chapter, and the next chapter as well, the Wilson activity model (Wilson, 1964) will be used to illustrate the text. Code lines are typeset in a monospaced typewriter font to separate it from the rest of the text, typically like this: `RGrad::constructor`, where `RGrad` is the name of a Ruby module that mirrors the name of the language. The constructor mirrors what type of object is created.

To keep the number of code lines at a minimum, and to make the progress easier to follow, the code examples are split into small parts and spread out over the text where it is natural with an example. The code lines are in most cases placed before

their actual explanations to give a motivation for what is coming. Only a minimal amount of code to illustrate the point is shown; and therefore, to give the total picture, the code is shown in full in Appendix C.

7.1 Unit Consistency Check

The RGrad language described in the following sections has one important feature which must be properly understood: All the programming elements (variables, parameters and expressions) created within the language have scientific units defined. As a starting point all the basic elements of the model, that is the variables and parameters, must be given the correct scientific units by the user. For all the expressions defined within the model closure, the units are calculated based upon the units of the elements and the λ-function given. The units of the expression are checked in full each time a new expression is created, and an error message is raised if the expression is found to be inconsistent. There is no way to check that the basic elements are given the correct units, but since these elements, and the expressions created from them, are used throughout the entire model it is virtually impossible to create an inconsistent model. It is also possible to check that certain derived expressions, such as for instance pressure or chemical potentials, have their expected units when the model is exported. The consistency check is also propagated through the gradient calculations and the Legendre transforms. If all the derived expressions are created without an error, it is a very good indication that the model is consistent. The consistency check does of course not guarantee that the implementation is correct, but it helps avoiding some very nasty errors. The unit checking also turned out to be a very good tool for debugging the code, since the location of the error is pinpointed by the consistency check.

In the gradient calculations it is possible to utilise the consistency check more extensively than for user defined expressions. The units of the gradient of the expression can be calculated based upon the units of the original expression, and the units of the element for which the gradient is calculated. If this implicitly derived unit does not coincide with the automatic gradient calculation, an error is raised.

The units that are given for the base elements, and those calculated for derived elements of the model, are transmitted to the exported code as well. Here the units are used to scale the model in an easy and straightforward way, see Section 8.6.13 on page 147. Since all the units are based on a small set of base units chosen by the user, it is, in order to scale the complete model, enough to change the base units. This will in turn affect the numbers given to the model, and also the numbers

taken out from the model. Internally, however, the computation is unaffected by the scaling.

It is not possible to turn off the consistency check, but if no units are given to any of the elements used in the model, the model is taken to be consistent. This means that the model is dimensionless and that scaling has no effect.

7.2 User Interface

The RGrad language is used to capture all the user information into the output language, and it must therefore reflect the output language in many ways. The internal representation therefore focuses on transmitting from the input to the output language, and is not intended for human interaction. This makes of course debugging a little challenging. To aid debugging some methods were implemented to dump intermediate results to the screen. These methods are not discussed here, but the internal representation is discussed in Section 7.3.

The RGrad language implements the grammar as it is described in Chapter 6, but it also relies heavily on the Ruby programming language (Thomas et al., 2005). The RGrad language uses the Ruby parser instead of implementing its own parser, and simply adds the needed functionality on top of the Ruby language. This gives the advantage of the full Ruby language, but at the same time it restricts RGrad to comply with the Ruby syntax which, however, is quite rich and flexible. Without utilizing a high-level language such as Ruby to write the input language this project would not have been possible within the given time frame.

A big effort has been made to write the constructor calls of RGrad as uniform as possible. Only the basic information is given in the constructor call, and all optional information is supplied in method calls afterwards. This gives the flexibility to add information to objects at a later stage, but it is nevertheless recommended to give as much of the information during construction. This will make the code easier to read, and it will prevent unexpected side effects, such as calculated results not being visible in the exported code because the label was given too late, or an expression is mistakenly said to be inconsistent because the units were not given in time. Information added after an object is used might be ignored without warning, and information given in the wrong order might raise an error. If for instance a dimension is used to define a variable or parameter, and this variable or parameter is instantiated before the size of the dimension is given, the program will raise an error because the number of elements does not match with the size of the dimension.

Units are not mandatory, but if the units are defined for one object, then they must also be given to all other objects of the model to make it consistent. The consistency check is done alongside with the construction of the expression. This means that the units must be given before the object is used in an expression. It is not possible to give units to an expression, as these should be calculated solely from the objects that are part of the expression, and nothing else.

The objects that are used to describe the model will to a large extent be interconnected. This means that all expressions depend on simpler expressions, which finally depend on the parameters, and the variables. Figure 6.3 on page 83 illustrate the pattern. This means that all the objects needed to calculate the result of an expression can be reachable from within the object. This situation can be represented in a graph, where the root node of the graph is the final result of the calculation.

```
rgrad = RGrad::graph
rgrad.set_timestamp("2008-01-01T00:00:00Z")
rgrad.set_helpstring(%Q{\
This is an implementation of the Wilson model.
The model is differentiated with respect to t and n.
This is Legendre transformed with respect to n.\
})
```

Listing 7.1: The graph is created, and the time stamp and the help string are added.

A particular model description can, and in most cases will, hold more than one root node, and a way to collect all the root nodes is therefore needed. These root nodes are collected in a graph. The graph object will in the end hold the complete model, with all the gradients, the Legendre transforms and other things that are needed. Note that only the root nodes need to be added to the graph. The derived objects can be stored as local variables, but as long as they are used, directly or indirectly, in an expression which is added to the graph, these objects will be included automatically. Once the model implementation is completed, and all objects are added to the graph, the model can be finalized. This process removes all duplicate objects from the graph and makes further changes to the graph impossible.

For run-time purposes the graph is equipped with the possibility to add two information strings, which are useful after the code has been exported. The time stamp is used to recognize when the model was written. This makes it possible to distinguish two models with equal file names, but are created at separate times. The format of the time stamp must follow the ISO standard time format, as shown above (ISO 8601:2004, E). The intention of the help string is to enable the programmer of the model to add a formatted string which tells the user what the model is all about, and how the API looks for instance in Matlab or Ruby.

It is important to note than the arrangement order of the different elements of the model is determined solely from the order of when the objects are used for the first time, and is thereby not necessarily in the order they are created. This means that the objects are not exported in the same order as they are written. This does of course not affect the numerical results.

7.2.1 Dimensions

```
nc     = RGrad::dim.init(3)
```

Listing 7.2: A dimension is defined and initialized.

The dimension objects are used to determine the size and shape of the objects in the model. A dimension object makes sense only in combination with other objects. For example: The structure of a model is usually given by the number of chemical components in the system and this should be be reflected by a dimension object. Each unique dimension should have a dimension object, but on the other hand the dimension objects should also be reused when dimensions are known to be equal*. It is also possible to have sub-models with a number of components which is different from the main model. This will of course require more dimension objects.

7.2.2 Variables and parameters

```
t      = RGrad::var.init(298.15).units!(
     {"unit_temperature"=>1}).label('temperature')
n      = RGrad::var(nc).init([0.2, 0.3, 0.5]).units!(
     {"unit_amount"=>1}).label('mole_numbers')

RGAS = RGrad::fix.init(8.3145).label("fix_rgas").units!(
     {"unit_time"=>-2, "unit_length"=>2, "unit_amount"=>-1,
      "unit_temperature"=>-1, "unit_mass"=>1})
```

Listing 7.3: Variables and parameters are created and initialized. A dimension object is used to tell that the variable n has one dimension. The numbers given to the units are the exponents.

There are four different ways of creating elemental objects within the RGrad language: One way of declaring variables, and three different ways of declaring parameters. The reason for dividing parameters into three different types has to do with flexibility and optimization of the exported code. Some parameters are universal constants and should not be changed, while others are experimentally determined model parameters. This means that some parameters should remain

*Equal both in size and interpretation

constant, while others could change in the future, based on new measurements or parameter optimization. This opens up for a convenient flexibility in the exported code, see also Chapter 8.

The constant parameters can again be divided into two different types; constants with a physical interpretation, and numeric model coefficients. The physical constants often have units, and this information must be given in order to maintain a consistent model. The physical constants do also retain a value outside the model context, and it is desirable for the end user to get access to these constants when using the end product. Typical examples of such physical constants are the gas constant, Avogadro's number etc.

Numeric model coefficients are typically exponents in polynomials and other constants originating from the derivation of the model. These model coefficients have no meaning outside the context of the model, and is of no interest to the end user. These coefficients can therefore be implemented as numbers directly into the λ-functions, and are as such indistinguishable from any other numerical values. The physical constants can also be used directly inside the λ-functions, but the extra information is stored in the computer graph, and a copy of the value is therefore accessible from the outside afterwards, provided that the constant is given a label. What happens further depends on the output. It is possible to implement all constants as ordinary parameters, but incorporating them directly into the λ-functions yields significantly faster code because the numerical value of each constant is compiled directly into the function, and is not given through the function header.

Another issue that separates real parameters from constants, is that the real parameters can have dimensions. Dimensions can only be given as arguments to the constructor, and cannot be changed afterwards. However, by changing the broadcasting, the interpretation of the dimension can change, see Section 6.4.

Parameters and variables have an optional initialization method where default values can be inserted. These values are stored inside the parameter object in a flat array, but the user is free to use sub-arrays in the constructor for increased readability. The number of values must correspond to the number calculated from the dimensions of the object, but the structure of the argument will be ignored. If the total number of input values are different from the number calculated from the dimensions, an error is raised. Variables are treated in the same way as ordinary parameters, and dimensions are given as arguments to the constructor. The default value of parameters and variables is 1.0 when the object is exported this value will be used if no other values are given.

Finally, each object is given an internal serial number, which is guaranteed to be unique. These automatically generated internal names make it difficult for the user

to recognize the model elements afterwards, and it is therefore possible to supply user defined names as well. It is legal to give more than one name to the same object. It is, however, important to note that the same name should not be given to two different objects. This leads in fact to an erroneous code. There is no check to make sure that this does not happen, and it is therefore up to the user to avoid name clashes*. The label given to an object will be added to the list of names already given, which means that not all the labels need to be given at the same time. As the model is built up the objects are copied, but the added labels will affect all the copies of the same object.

7.2.3 Expressions

The Ruby language gives the possibility to attach an anonymous function, called a block, in the constructor call. See Thomas et al. (2005) for further details about blocks. Listing 7.4 shows a typical example of how expressions are constructed.

```
ntot   = n.sum
lm     = RGrad::expr(lij,n[nil,nc]){
    |_lij,_mj| _lij*_mj
    }.sum!(nc,nil)
llm    = RGrad::expr(lij,n[nc,nil],lm[nc,nil]){
    |_lij,_mi,_lmi|_lij*_mi/_lmi
    }.sum!(nil,nc)
rgrad << RGAS
mu     = RGrad::expr(t[nil],ntot[nil],lm,llm){
    |_t,_ntot,_lm,_llm|RGAS*_t*(1-(_lm/_ntot).log-_llm)
    }
euler = RGrad::expr(n,mu){
    |_n,_mu|_n*_mu
    }.sum!.grad!(n,mu)
```

Listing 7.4: Expressions are defined, and different uses of summation is shown. The names given between the pipes (|) define the names of the local variables in the λ-function, which is defined afterwards. The hand coded gradient, *mu*, of *euler* with respect to *n* is defined.

The objects that are used in the expression must be given as a list of arguments to the constructor. The arguments can be a mix of variables, parameters and other expressions, as illustrated by Listing 7.4. The sizes of these arguments do not need to be the same in the first place, but they must conform as they enter the expression. This is done by a technique called broadcasting, see Section 6.4 on page 86. By definition, the objects which already has the correct size do not need to be broadcasted. Broadcasting in itself changes the object that needs to be broadcasted, i.e.

*When the final code is produced, most compilers will issue a warning that a constant has been redefined.

a copy of the object is used, and the dimensions are augmented during the broadcasting. The broadcasting will only make sense in the context of the other objects given as arguments to the same expression. The broadcasting is specified entirely by the user, and the user is responsible for making all the objects of the expression conform. If this is not the case, an error is raised. When broadcasting is used, the full rank of dimensions must be given, and the order of the original dimensions of an object cannot be changed. When a dimension needs to be broadcasted, a `nil` is given for this dimension. If a dimension is not broadcasted, the corresponding dimension object is given. When broadcasting is used, the dimension is implicitly determined by the dimensions given to the other objects of the expression. If broadcasting by mistake is used in the same position for all objects given to the expression, it is impossible to determine the dimension, and an error is therefore raised. For more details about broadcasting, see Section 6.4 on page 86.

The dimensions of the new expression are calculated based upon the objects which are given to the constructor. Interestingly these dimensions can be overridden by the user. This will typically happen when the user manage to reduce the dimension of the expression by knowledge of the problem. This will reduce the amount of memory needed for this expression, and it will calculate the result faster. The correct dimensions are set afterwards, to make sure the rest of the code is not broken as a result of this optimization. This will ensure that the reduced dimension is broadcasted when this object is used later.

There is a hidden link between the objects in the list given as an argument to the expression and the objects given to the λ-function given: The order of the objects given in the list, and the corresponding objects given in the λ-function must be equal. It is up to the programmer to make sure this is the case, and a failure to do so will produce erroneous results. There is no way to make sure this is correct, but the units check explained in Section 7.1 on page 92 will in most cases handle these errors. An error is raised if an illegal object type is given in the λ-function. The only objects allowed in the λ-function are local variables representing scalar values from the objects in the list, the constants defined with or without units, and numerical values.

There are two different ways of using a sum in combination with an expression. Both give the same result, but one require more memory than the other. The first type of sum (`sum`) is used to sum the result of the expression. This will be the same type of sum which can be performed on variables and parameters. The other type of sum (`sum!`) is performed while the expression is calculated. Both sum methods uses the same notation as the broadcasting to indicate which dimensions to sum, and which to keep. The number of dimension objects given to the sum must comply with the number of dimensions in the expression, otherwise an error

will be raised. There is one exception though: When no arguments are given it is tacitly assumed that all the dimensions are summed. The second type of sum is used in exactly the same way, but internally, `sum!` is performed while calculating the result. This saves memory in the computation code, since it is not necessary to allocate memory for the dimension which is contracted. The `sum!` method is used when only the summed expression is needed. The method `sum` should be used if both the summed expression and the parent expression are needed in the model.

```
flg = RGrad::flag.label('flag_2_derivatives')
rgrad<< RGrad::goto(flg).label('goto_2_derivatives')

rgrad<< RGrad::flag.label('flag_0_derivatives')
rgrad<< euler.label('g')
rgrad<< RGrad::flag.label('flag_1_derivatives')
rgrad<< euler.grad(t).label('g_t')
rgrad<< euler.grad(n).label('g_n')

rgrad<< flg
rgrad<< euler.grad(t).grad(t).label('g_tt')
rgrad<< euler.grad(n).grad(n).label('g_nn')
rgrad<< euler.grad(n).grad(t).label('g_nt','g_tn')
```

Listing 7.5: The `flag` statements are used to enable the execution to stop before the first or second order gradients are calculated. The `goto` statement is used to jump straight to the execution of the second order gradient.

The expressions interact with other expressions, variables and parameters to create new expressions. At any point it is possible to ask for the gradient of an expression and use this in a new expression. It is in principle possible to calculate the gradient with respect to any object known in the context, but for most cases it is only of interest to find the gradient with respect to variables, and maybe parameters. When higher order gradients are needed, the gradient method can be called on the object returned from another gradient method. The gradient created can be used as input to new expressions later on.

It is also possible to circumvent the gradient calculation and implement the gradient by hand. The gradient can then, and should be, registered inside the corresponding object by using the `grad!` method. This will override the automatic gradient calculation, and the hand coded version will be used. This is preferable when knowledge of the model enables the user to do simplifications which the automatic method would miss. All gradients are calculated only once for each object, so in this case the automatic gradient will never be calculated. Further calls to the `grad` method will then use the hand-coded version as the point of departure.

The automatic gradient is calculated for all child objects as well, as these are implicitly needed to calculate the final gradient. See Section 6.3 on page 84 for more

details. Hand coded gradients, however, does not require to calculate the gradient for the child objects.

Finally, there is an additional feature called Legendre transforms available to the expressions, see Sections 2.1 on page 7 and 7.5 on page 107 for more details.

```
leg = euler.legendre(n).label('z')
rgrad << leg
rgrad << leg.grad(n).label('z_n')
rgrad << leg.grad(n).grad(n).label('z_nn')
```

Listing 7.6: The expression *euler* is Legendre transformed with respect to n, and both the first and second order gradient are calculated.

The method that calculates the Legendre transform needs to know both which variable to transform, and what variables the transformed expression should be differentiated with respect to. The reason for this is explained in more detail in Section 7.5 on page 107. A transformed expression can be transformed repeatedly, but with a different variable each time. Again as explained later, reverse Legendre transforms are not possible because the interpretation of the variable will be internally wrong. No error will be raised, but the result will be erroneous. The gradient, however, can be calculated for the transformed expressions with the same method as for ordinary expressions.

The argument to the gradient method of the transformed expression is the same variable as for an ordinary expression, but if the expression is transformed with respect to this variable, the interpretation will be different. Unfortunately this is not transparent in the interface. The actual transformed variable is never created as such. The untransformed variable is used instead, and the situation where it is used determines whether the transformed or untransformed variable should be used. This interpretation happens in the background, and unfortunately it will not be transparent for the end user. If for instance $A(T,V,\mathbf{n})$ is transformed with respect to T, and the result $U(S,V,\mathbf{n})$ is differentiated with respect to T, it is actually differentiated with respect to the conjugate variable of T, which is $-S$ (see table 2.1 on page 12). This also explains why reverse Legendre transforms are not possible. The internal interpretation of the variables will be wrong. As an example, if A is transformed to U, the reverse transformation can be written: $A = U - \frac{\partial U}{\partial -S} - S$, where the variable S is only implicitly known. In this context the explicit variable will be T, which means that the last term in the expression will be wrong. When $\frac{\partial U}{\partial -S}$ is calculated, however, T will be correctly interpreted as $-S$. Unfortunately there is no general way to determine whether a variable should be interpreted as the original variable or a conjugate variable outside the proper context.

7.2.4 Control statements

```
rgrad  << RGrad::halt(n){
    |_n|_n
    }.min!.label('halt_negn')
```

Listing 7.7: This `halt` statement will calculate the smallest value in n and stop execution if any value is less than zero.

Thermodynamic models are never valid for the whole domain of numerical values. For instance, the logarithm of a negative mole number might yield a complex number. It is desirable to be able to catch these errors as soon as possible, and in such a way that it will be possible to correct the mistake at the user level. Such nonphysical results can for instance be the result of an iteration procedure taking a too aggressive step, or when asking for a step outside the physical region of the model. By enabling feedback to the user algorithm it is possible to try again with a reduced step size. As a consequence of this, it is possible to implement tests which will give a controlled exit of potentially erroneous calculations.

These tests are called `halt` statements. They are used much in the same way as expressions, but with a few important exceptions. A `halt` statement should always evaluate to a scalar value. If this value is less than or equal to zero, the test fails, and the execution stops. If the value is greater than zero, the computation continues. If the elements are not scalar, a method for finding the smallest value in the resulting structure can be called. To be able to recognize that a test has failed, these objects must be given a label. Even though the label is strictly required, it is given in the same lazy way as for all other objects. If no label is given an error is raised, but not until the object is exported. For the `halt` statements to be effective, the tests must be added to the graph before the rest of the code is calculated. This is achieved by adding the `halt` statements to the graph before the model expressions.

It is also possible to stop the execution at different locations specified by the user, for instance after the first order gradient has been calculated and before the second order gradient is calculated. This is achieved by creating an object designed for this purpose, called a `flag`, and this is added to the graph at the desired location. This gives flexibility to run only parts of the exported code. As for the `halt` statements, these statements require a label. To make the notation uniform, the label is again provided in a method call after the object has been created. The label is mandatory, since it must be possible to determine where the execution should stop in the exported code.

The final control statement is the `goto` statement, which enables execution jumps in the code. A `flag` can for instance be used to first calculate the first and second

order gradient only, and then afterwards a `goto` statement is used to calculate the third-order gradient without recalculating the first and second order gradients. For a `goto` statement to be effective, it should be added to the graph as early as possible in order to avoid unnecessary computations when a execution jump is made. The `goto` statements only works in combination with a `flag`. This means that both a `goto` statement and a `flag` must be added to the graph, and the corresponding `flag` must be put on the graph where the desired execution should start. The `flag` object is given to the `goto` constructor, such that this object knows where to jump. It is up to the user to put the `flag` at the correct location. Failure to do so will render inappropriate code, and unexpected results may occur.

7.3 Internal Representation

This section describes the key features of the internal representation of the objects in the RGrad language. All classes of the internal representation are protected by a single name space RGrad, and should only be accessed by class methods. This is purely a pragmatic convention to avoid name clashes.

Names, both internal (Ruby) and external (C), are generated by a name factory, which basically increments the name each time the method is called. This guarantees a unique naming scheme. As a design choice in the C-code described in Section 8.1 on page 118, all elements are stored in arrays, and the name of each element reflects the position in this array. Because of simplifications and optimizations made to the code, some of the created objects may be removed from the final code. All objects are therefore given a temporary internal name to avoid a sparse array. The internal name is used only to separate different objects from each other. After simplifications and optimizations the temporary name is replaced by a name which reflects the position in the array.

When a new expression is created, its dimensions are calculated based on the dimensions of the objects included in the expression. The list of dimensions should be dense, which means that no broadcasting is possible in the created object. If this is not the case, an error is raised. At the same time, the dimensions of the arguments are checked. All objects in the expression must have the same number of dimensions, and the corresponding dimensions should be equal, if not they must be properly broadcasted. An error is raised if this is not true.

The units of the expression is calculated based upon the input arguments, and the λ-function. At the same time, the expression is tested for consistency. An error is raised if the expression has inconsistent units. If the units are approved, they will

be stored inside the object, and the possibility to change them is removed. This is done to make sure that the units are not changed by a mistake. In practice the units checking has proved to be of immense value, as errors in the implementation were pinpointed at an early stage, actually before any calculations were made.

The λ-function of the expression must be converted to an internal format before it can be differentiated and exported. This is done by calling the function with input objects designed specially for this purpose. The input objects have certain properties to facilitate differentiation, and there is a link between the input objects and the corresponding objects in the argument list of the expression. The objects must also have a name that is used in the exported code. Since all λ-functions should have a uniform API, it was chosen to use an array as the only argument to the function internally. The name of the variables in the λ-function reflect this by behaving like an array dereferencing. The final information given to the input objects of the λ-function is the units of the corresponding objects in the argument list of the expression. One input object is created for each object in the argument list, and the λ-function is called. This returns a tree object which can be manipulated by the differentiation algorithm. It is of course important that the order of the objects given to the λ-function corresponds with the order the objects inside the argument list. If the ordering is wrong, the λ-function will calculate something else than what is intended. Unfortunately there is no way to check this, but since the objects have a fixed ordering, this should always be true, provided the user has not made a mistake.

7.4 Automatic Gradient Calculations

The generation of gradients is the most complicated operation which is undertaken on the expressions, see Section 6.2, and in order to save computation time each gradient is therefore computed only once. If the same gradient is required several times, a copy of the previously computed gradient is returned. As explained in Section 6.2, the chain rule requires that the λ-function has been differentiated with respect to all the variables, which in turn will generate a new expression for each derivative not evaluating to zero. The new expression must again be multiplied by the corresponding gradient. There will be a sum of multiplications, which in turn will give a new expression which represents the gradient.

The objects given as arguments to an expression have a corresponding element in the λ-function. The object from the list of arguments and the corresponding object in the λ-function are differentiated simultaneously. The λ-function is differentiated with respect to the variable corresponding to each element, and the gradient

is calculated for that element. If the gradient of the element exists and the differentiation of the λ-function is not zero, a new expression will be constructed. The expression looks is determined by the outcome of the differentiation of the λ-function, and there are two different alternatives for differentiating the λ-function, either internally or by calling external programs. The internal method will work directly on the internal representation of the λ-function and produce a new object in the same representation.

However, since the λ-function is a symbolic algebraic expression, it makes sense to differentiate this with a Computer Algebra System (CAS). The possibility of using an external CAS has therefore been implemented as an alternative to the internal differentiation. In principle any CAS which takes command line arguments can be used for the purpose. The only requirement is to translate the internal representation to a string which is read by the CAS, and interpret the result afterwards. This has been demonstrated for two different computer algebra systems, namely Maple and Ginsh. The latter is a front-end to GiNaC (Bauer et al., 2002). The great advantage of these programs is that they render expressions with fewer operations than the internal differentiation*. The numerical result are the same within the machine precision, but with fewer operations, and therefore the final code will run faster. The disadvantages are mainly compatibility issues, and all external programs require some kind of license, either commercial (Maple) or not (Ginsh). Another great disadvantage is that the overhead in making these external calls significantly slows down the gradient calculations. This problem can probably be solved, or at least reduced by linking the computation kernels into Ruby. This should improve the computation time significantly since the external library is then loaded only once per run instead of once per differentiation.

A new expression based on the differentiated λ-function must be created unless the differentiation evaluates to zero, or the gradient of the corresponding object from the argument list does not exist. The new λ-function raises three different possibilities: The λ-function can result in another λ-function, the λ-function may reduce to one single object, or the λ-function may reduce to a numerical value.

Whenever a λ-function is differentiated, and the number of variables in the derivative is decreased compared to the parent function, it means that some of the corresponding objects in the argument list are not needed in the argument list given defined for the differentiated expression. The differentiation can also possibly change the order of the variables. To remove any obsolete objects, and to figure out the new argument order, the new λ-function is traversed in the order it is evaluated,

*This is always the case for Maple with optimization option tryhard. It is, however, not always true for Ginsh or Maple without optimization. The reason for this is probably that CAS focuses on generating human readable code. This is not always the fastest computation wise.

and each unique object that appears in the λ-function is collected. When this operation is complete, the corresponding objects are picked from the argument list, and a new argument list is created without obsolete elements, and with the correct ordering.

If the traversal of the λ-function renders an empty argument list it is either an error, or the λ-function has reduced to a numerical value, most likely a fraction. The reason that the fraction appear is that the internal representation uses operator overloading and lets Ruby handle numerical operations, with one important exception: Integer fractions will be truncated by Ruby, for instance 1/2 will become 0. In order to avoid truncation errors internally, these divisions are not converted to floating point numbers before the expression is exported. This special case is treated in exactly the same manner as when an ordinary numeric value is the only object which is left after differentiation. It does not make sense to create a structure to hold one scalar value, and maybe copy this into several dimensions. Scalar values should therefore be integrated into the λ-function. The only problem is that the λ-function is not generated yet. To solve the problem, a temporary object is created, and the scalar value is substituted into the λ-function when this is created.

A new expression is created based on the new λ-function and the original expression in a case where the derivative of the λ-function is not a numeric value. As already mentioned, a new argument list is created based on the new λ-function and the original argument list.

The objects in this list must be copied from the original expression to make sure the originals are not destroyed. The broadcasting of the objects are kept when they are copied, but the broadcasting is not necessarily correct in the setting of the newly created expression and must therefore be adjusted. The broadcasting must be adjusted if the differentiation of the λ-function results in the removal of objects from the argument list. This means that it might be possible to reduce the number of dimensions of the new object compared to the original. This will save both computation effort and memory in the final code. If a dimension is removed from the new expression, this dimension must be replaced by broadcasting when the expression is used. According to the chain rule, the new expression must be multiplied with the corresponding gradient, as explained in Section 6.3 on page 84. It basically means that the broadcasting of the new expression might need to be adjusted such that this multiplication is possible.

The gradient calculation of an expression can be shown with the following example:

```
e = RGrad::expr(x[d, nil], a, y[d, nil]){|_x, _a, _y|
    _x*_a*_y}
```

The symbols x, a and y represent any differentiable objects. Their gradients are assumed to be known and are here called gx, ga and gy respectively. The gradient of e will create a total of four new expressions, which are shown below. The objects dx, da and dy are due to the differentiation of the λ-function which must satisfy the chain rule.

```
dx = RGrad::expr(a, y[d, nil]){| _a, _y|
    _a*_y}
da = RGrad::expr(x, y){|_x, _y|
    _x*_y}
dy = RGrad::expr(x[d, nil], a){|_x, _a|
    _x*_a}
```

Notice that the broadcasting of da has been removed, and that the reduction in rank is compensated for when the expression is used to calculate the gradient of e:

```
ge = RGrad::expr(gx[d, nil, d], dx[d, d, nil], ga, da[d, nil, nil],
                 gy[d, nil, d], dy[d, d, nil]){
                 |_gx, _dx, _ga, _da, _gy, _dy|
                 _gx*_dx + _ga*_da + _gy*_dy}
```

The last option for the differentiation of the λ-function occurs when it is reduces to a single object. When this happens it is not necessary to create a new object, since a copy of the remaining object can be used instead. The broadcasting of the new object must be updated to the new context of course, and the broadcasting must be augmented according to the dimensions of the arguments given to the gradient method.

The new object, copied or created, is matched with the corresponding gradient. Since the gradient of an object is calculated only once, the dimensions with broadcasting must be reset in the context where the object is used. Once all gradient-expression from the differentiated λ-function pairs are collected, a new expression representing the gradient can be constructed. The λ-function for this expression is constructed based on these collected objects. As explained in Section 6.2, this λ-function is really a sum of products. This issue is solved by using arrays to collect the objects on two levels. The objects at the first level are added, while the objects at the second level are multiplied. At the same time the function is simplified by inserting scalar values directly into the λ-function, as explained earlier.

The expression representing the gradient must be summed over the same dimensions as the original if the original expression was a summed expression. If the number of dimensions for this new expression has increased compared to the original expression, the broadcasting of the sum must be adjusted accordingly.

Every time a gradient is calculated with respect to a new variable, the differentiation of the λ-function will render the same expressions. It therefore makes sense

to create these expressions once, and only use copies later. This approach was attempted, but generated a bug in the broadcasting which could not be resolved at the time. It was therefore decided to implement a method which could recognize these duplicate expressions, and replace them afterwards. What is important with this procedure is that the exact same result is not calculated more than once in the final code. Several copies of the same object will appear in the model graph, and when these objects are exported, the unique objects are distinguished by comparing the names of all the objects, see Chapter 8 for more details. It is therefore sufficient to change the names of the duplicate expressions that are created, to make sure the expression is calculated only once. A string representation is used to compare the different expressions with each other. This string is based on the argument list, the broadcasting and the λ-function of the expression, and will therefore be unique for all unique expressions. When two identical expressions are found, the name of the second is simply replaced by the name of the first one. This will automatically affect the name of all the copies of the object as well, and it will also affect the string representation of the expressions which depends on this object. This means that in order to be sure that all duplicate expressions are removed several passes must be run. In practice the optimization is run in an indefinite while loop until no more expressions are replaced. In practical calculations it is normally enough with three to four passes, but up to eighteen passes has been registered.

7.5 Automatic Legendre Transforms

The theory in Chapter 2 gives the basis for implementing an automatic Legendre transform. The Legendre transform is in itself relatively straightforward, and the major challenge lies in generating the gradients. The gradient of the Legendre transform is also needed in order to transform more than one variable set. Note that any gradient of the Legendre transform depend on the gradients of the untransformed function to the same order of differentiation. The gradient operations are easily achieved by the procedure described in the previous section.

It is certainly desirable to generate the result within the grammar described in Chapter 6, but this is unfortunately not possible because the gradient of the Legendre transform requires the inverse of a matrix. The grammar has no concept of a (matrix) inverse, and it would be impossible to introduce a general inversion procedure into the grammar because of its multidimensional nature. The problem can, however, be avoided by generating the inverse outside the grammar, and then use the result in the context of RGrad. It must be stressed that this is a special operation provided in order to facilitate the gradient of the Legendre transform, and should

not be considered as a part of RGrad, or the grammar itself. The operation takes place in the background, and the inverse method will not be available to the end user. For practical purposes, the distinction is not visible in the implementation, but it requires a few hacks that are not welcome to the programmer.

The first and second order gradients do not fall into the same calculation scheme as the third and higher order gradients. This is because of certain simplifications, which indirectly affect the higher order gradients as well. The simplifications can potentially reduce the runtime of the final code by one order of magnitude. Since these simplifications are so important, and the number of expressions is quite low, all the first and second order gradients are always calculated, regardless of what the user actually needs. The first argument to the `legendre` method is the variable to transform, and because of the pre-calculation of the first and second order gradients, all variables that are used in the gradient calculations at a later stage must be given as arguments as well.

Even though the first and second order gradients are made ready in the background, they are not added to the graph, which means the code required to calculate these gradients is not generated unless the user explicitly asks for it. The transformed expression behave just like ordinary expressions, with one important exception, namely that it cannot be transformed with respect to the same variable again. Inverse Legendre transforms are therefore not possible. Transforming a Legendre function twice with the same variable, should give back the original function but this will not happen here. The reason for this is that the variable transformations internally are based upon the original variables only. The Legendre transform is required to find the gradient of the function with respect to the transformed variable. This gradient is multiplied with the variable. In this case, the gradient calculation will work, but the variable will not be the transformed variable, thereby generating erroneous results. Unfortunately it is not possible to prevent this from happening within the existing code, see Section 7.2.3

$$
\begin{aligned}
\text{Rule 2.23: } g_{\sigma\sigma\sigma} f_{rr} &= g_{r\sigma\sigma} \\
g_{r\sigma\sigma} f_{rr} + g_{\sigma\sigma} f_{rrr} &= 0 \\
g_{\sigma\sigma\sigma} f_{rr} f_{rr} + g_{\sigma\sigma} f_{rrr} &= 0 \\
g_{\sigma\sigma\sigma} f_{rr} f_{rr} - f_{rr}^{-1} f_{rrr} &= 0 \\
\text{Combine and order: } g_{\sigma\sigma\sigma} &= f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1}
\end{aligned}
$$

The equations shown above comes from Appendix A, and show that the matrix inverse f_{rr}^{-1} that comes from solving a system of linear equations. This system of

equations is lower triangular, and is therefore quite trivial to solve, but is impossible to calculate the gradient analytically because of the inverse. Implicit differentiation must therefore be used, and the system of equations must then be solved afterwards. This gives some extra challenges in creating an algorithm for providing the gradients. The solution is to avoid solving the expression before it is exported. By doing so the inverse does not make problems in the differentiation. It is, however, important to keep track of the expression, such that solving it afterwards does not pose a problem. The implicit expressions are not created because they have a simple structure and is therefore easy to solve. There are no intentions of implementing implicit expressions to RGrad at this point; therefore the expressions created here are treated with a special code made for the sole purpose of generating gradients of Legendre transforms.

By looking at the expressions derived in Chapter 2 and Appendix A, the expressions are seen to be divided into three different parts. The first part is the term which is multiplied by the solution of the implicit expression. This is either the number 1, in which case the solution is trivial, or the second order gradient. This is a symmetric matrix with the same size as the element used in the differentiation. The rest of the expression is ordered by the sign of the elements, and divided into two parts. All terms with the same sign as the main term will, get a negative sign when the expression is solve, whilst the other terms will get a positive sign.

The three parts are each stored in arrays. The first array will at most include one object, but it is nevertheless given the same structure as the rest of the arrays. The terms involved in the expression are either a sum of different terms, or an inner product between two, and only two, elements. Since the expressions have such an easy structure, a simple procedure using sub-arrays is implemented to collect the terms of the expression. The objects on the first level of the array are addition. On the second level, there will be an inner product. This structure is also maintained during differentiation. As an example, the expression

$$f\alpha = \beta + \gamma - \delta - \varepsilon\varphi \tag{7.1}$$

is stored this way:

$$\begin{aligned}&[\alpha]\\&[\beta,\gamma]\\&[\delta,[\varepsilon,\varphi]]\end{aligned}$$

Parallel to these arrays, expression equivalents are generated and stored inside the object. Because the derivation of the gradients is different for Legendre transformed expressions, these expressions are used only when output is exported. For

each Legendre transformed expression it may be necessary to create several ordinary expressions. The reason is that each inner product must be calculated before the result can be added to the rest of the terms in the expression. Creating these inner products is not straightforward, but since all the elements involved in the expression are created internally, the number of outcomes is kept at a minimum. The two elements in the inner product have only one non-broadcasted dimension in common. This is the dimension which the sum will be calculated over. The dimension is determined by searching through the dimensions of the two elements. There is also the possibility that the inner product is a scalar because the sum given by Rule 2.23 may have no dimensions, and consequently there will be no sum. The λ-function is in this case relatively simple, and involves one single multiplication only. All implicit equation must be solved in order to construct the final expression. Instead of broadcasting every term in the expression an extra expression is constructed. In this expression, the λ-function will consist of only plus and minus, depending on which side of the equal sign the term was in the original expression. All The terms are collected accordingly in two separate arrays, and it is relatively easy to construct the λ-function and the expression. The broadcasting of this expression must finally be adjusted to fit the inner product with the inverse in order to complete the final expression.

Since all the expressions are made simultaneously, it is possible to assure that the units of the gradients of the Legendre transform are consistent, and possible errors are given in the proper place, see Section 7.3. It is also possible to assure that the dimensions and broadcasting are correct.

7.5.1 Calculating gradients of Legendre transforms

The interface to create gradients of the Legendre transformed expressions is the same as for ordinary expressions, but the interpretation is not the same. As mentioned earlier, the first and second order gradients are implemented by hand, while higher order gradients follow the scheme derived in Chapter 2:

$$\begin{aligned} h_\sigma f_{rr}\,\mathrm{d}r &= h_r\,\mathrm{d}r \\ h_\theta\,\mathrm{d}t + h_\sigma f_{tr}\,\mathrm{d}t &= h_t\,\mathrm{d}t \\ h_r &= h_\sigma f_{rr} \\ h_t &= h_\theta + h_\sigma f_{tr} \end{aligned}$$

The two different types of expressions originate from Rules 2.23 and 2.24 which are repeated above, and which in turn means that two different methods are re-

quired for constructing the gradient. An expression can only be transformed with respect to one variable at a time. This means that if the gradient is calculated with respect to the transformed variable, one method is called, while a different method is needed for gradients with respect to untransformed variables. These methods do only apply Rules 2.23 or 2.24. There is a different method for the actual gradient calculations. The main differentiation method will successively go through the three arrays within the object, and create new objects of the correct type. The structure described in the previous section is used both to differentiate, and to construct the new objects. Since the expression is implicit, the differentiation will also be implicit. If the first term of Equation 7.1 is present, differentiation gives two new terms according to the product rule of differentiation. As an example the third-order gradient of U created from A by transforming T is shown below. The second order gradient of U with respect to $-S$ and V can be written like:

$$U_{-SV}A_{TT} = A_{TV}$$

This will be stored internally like:

$$[A_{TT}]$$
$$[A_{TV}]$$
$$[]$$

When U_{-SV} is differentiated with respect to V this will become:

$$U_{-SVV}A_{TT} + U_{-SV}A_{TTV} = A_{TVV}$$
$$[A_{TT}]$$
$$[A_{TVV}]$$
$$[[U_{-SV}, A_{TTV}]]$$

The first term is of the same kind as the original; while the second term is placed in the second part of Equation 7.1. Differentiation of the last two parts of the transformed expressions are straightforward. The only task to consider is to call the correct method for differentiating the transformed expressions.

7.6 Potential Pitfalls

In this section some potential pitfalls when models are implemented are discussed. The examples are nontrivial, and errors occur only when the models are differentiated, except for the misuse of goto statements. It is important to stress that these errors are not due to bugs in RGrad, but to assumptions made outside the realm of the RGrad language, or to errors made while the model is implemented.

7.6.1 Optimization side effects

When a model is first written, it may be possible to exploit favourable properties of the model, which is impossible to catch automatically. As will be shown here, this can have unexpected side effects. As an exercise the same equation is written and differentiated in two different ways. The first set of equations is written with no prior problem knowledge, while the last set of equations is written with the assumption that the coefficient array a is symmetric. This opens up for significant optimizations while implementing the gradient. In the equations below, the subscript to f means that it has been differentiated with respect to that variable. The bullets and open circles reflect the broadcasting explained in Section 6.4 on page 86. An open circle means that the corresponding dimension is broadcasted, and a closed circle means that the true dimension of the object is used. The gradient of f is derived with the gradient of x in mind, and as can be seen from the expansion below, the gradient with respect to a is wrong. This means that the assumption of a being symmetric is only valid when the expression is differentiated with respect to x.

Unsymmetric equations:

$$f = \sum_{\bullet\bullet} x_{\bullet\circ} a_{\bullet\bullet} x_{\circ\bullet} \tag{7.2}$$

$$f_x = \sum_{\bullet\bullet\circ} a_{\bullet\bullet\circ} x_{\circ\bullet\circ} \delta_{\bullet\circ\bullet} + x_{\bullet\circ\circ} a_{\bullet\bullet\circ} \delta_{\circ\bullet\bullet} \tag{7.3}$$

$$f_a = \sum_{\bullet\bullet\circ\circ} x_{\bullet\circ\circ\circ} x_{\circ\bullet\circ\circ} \Delta_{\bullet\bullet\bullet\bullet} \tag{7.4}$$

$$f_{xa} = \sum_{\bullet\bullet\circ\circ\circ} x_{\circ\bullet\circ\circ\circ} \Delta_{\bullet\bullet\circ\bullet\bullet} \delta_{\bullet\circ\bullet\circ\circ} + x_{\bullet\circ\circ\circ\circ} \Delta_{\bullet\bullet\circ\bullet\bullet} \delta_{\circ\bullet\bullet\circ\circ} \tag{7.5}$$

$$f_{xx} = \sum_{\bullet\bullet\circ\circ} \delta_{\circ\bullet\circ\bullet} a_{\bullet\bullet\circ\circ} \delta_{\bullet\circ\bullet\circ} + \delta_{\bullet\circ\bullet\circ} a_{\bullet\bullet\circ\circ} \delta_{\circ\bullet\bullet\circ} \tag{7.6}$$

$$f_{xxa} = \sum_{\bullet\bullet\circ\circ\circ\circ} \delta_{\circ\bullet\circ\bullet\circ\circ} \Delta_{\bullet\bullet\circ\circ\bullet\bullet} \delta_{\bullet\circ\bullet\circ\circ\circ} + \delta_{\bullet\circ\circ\bullet\circ\circ} \Delta_{\bullet\bullet\circ\circ\bullet\bullet} \delta_{\circ\bullet\bullet\circ\circ\circ} \tag{7.7}$$

Symmetric equations:

$$f = \frac{1}{2} \sum_{\bullet} x_{\bullet} (f_x)_{\bullet} = \sum_{\bullet} x_{\bullet} \Big(\sum_{\bullet\circ} x_{\bullet\circ} a_{\bullet\bullet}\Big)_{\bullet} \tag{7.8}$$

$$f_x = \sum_{\bullet\circ} x_{\bullet\circ} (f_{xx})_{\bullet\bullet} = 2 \sum_{\bullet\circ} x_{\bullet\circ} a_{\bullet\bullet} \tag{7.9}$$

$$f_a = \frac{1}{2} \sum_{\bullet\circ\circ} x_{\bullet\circ\circ} (f_{xa})_{\bullet\bullet\bullet} = \sum_{\bullet\circ\circ} x_{\bullet\circ\circ} \Big(\sum_{\bullet\circ\circ\circ} x_{\bullet\circ\circ\circ} \Delta_{\bullet\bullet\bullet\bullet}\Big)_{\bullet\bullet\bullet} \tag{7.10}$$

$$f_{xa} = \sum_{\bullet\circ\circ\circ} x_{\bullet\circ\circ\circ} (f_{xxa})_{\bullet\bullet\bullet\bullet} = 2 \sum_{\bullet\circ\circ\circ} x_{\bullet\circ\circ\circ} \Delta_{\bullet\bullet\bullet\bullet} \tag{7.11}$$

$$f_{xx} = 2 a_{\bullet\bullet} \tag{7.12}$$

$$f_{xxa} = 2 \Delta_{\bullet\bullet\bullet\bullet} \tag{7.13}$$

Unsymmetric example $a_{12} \neq a_{21}$

```
f    = x1*x1*a11 + x1*x2*a12 + x1*x2*a21 + x2*x2*a22
```

```
fx   = [2*x1*a11 + x2*a12 + x2*a21,
        x1*a12 + x1*a21 + 2*x2*a22]
fa   = [[x1*x1,x1*x2],[x1*x2,x2*x2]]
fxa = [[[2*x1,0],[x2,x1]],[[x2,x1],[0,2*x2]]]
fxxa = [[[[2,0],[0,0]],
          [[0,1],[1,0]]],
         [[[0,1],[1,0]],
          [[0,0],[0,2]]]]
```

Symmetric example $a = a_{12} = a_{21}$

```
f    = x1*x1*a11 + 2*x1*x2*a + x2*x2*a22
fx   = [2*x1*a11 + 2*x2*a, 2*x1*a + 2*x2*a22]
fa   =[[x1*x1,x1*x2],[x1*x2,x2*x2]]
fxa  = [[[2*x1,0],[2*x2,0]],[[0,2*x1],[0,2*x2]]]
fxx  = [[0,2*a],[2*a,0]]
fxxa = [[[[2,0],[0,0]],
          [[0,2],[0,0]]],
         [[[0,0],[2,0]],
          [[0,0],[0,2]]]]
```

As shown by the two-dimensional symbolic expansion of the expressions, the gradient with respect to a is erroneous in the symmetric case. This example represents a typical part of a thermodynamic model. The desired result from these calculations is the gradients with respect to the variable x, while the gradient with respect to the symmetric parameter array a is not so commonly used. More importantly, when the gradients with respect to the variables are used, computation speed is more important than when parameter sensitivities are calculated. From the equations it is easy to see that the symmetric optimization will be faster than the expressions that are not optimized. The expression for f_x needs n times as many calculations in the unsymmetric case compared to the symmetric case. It is therefore recommended to implement such optimizations if possible, but the programmer must be aware that this could have bad side effects. If the gradient with respect to the parameter array is also needed, both versions should be implemented in parallel. As long as only f_x is needed the optimized code is used, while f_{xa} will activate the unoptimized code, which properly calculates the gradient with respect to a.

7.6.2 Gradient order

The differentiation order of partial derivatives of continuous functions is in general immaterial. Thermodynamic functions are for the most part continuous and differentiable, and it is therefore expected that the partial derivatives are independent of differentiation order, but this turns out to be true only for scalar derivatives. The expressions from the previous example can be used to show that this is not true in the multidimensional case even though there are only scalar operations in the λ-functions. By differentiating Equation 7.3 and 7.4 with respect to a and x respectively, we get:

$$f_{xa} = \sum_{\bullet\bullet\circ\circ\circ} x_{\circ\bullet\circ\circ\circ} \Delta_{\bullet\bullet\circ\bullet\bullet} \delta_{\bullet\circ\bullet\circ\circ} + x_{\bullet\circ\circ\circ\circ} \Delta_{\bullet\bullet\circ\bullet\bullet} \delta_{\circ\bullet\bullet\circ\circ} \tag{7.14}$$

$$f_{ax} = \sum_{\bullet\bullet\circ\circ\circ} x_{\circ\bullet\circ\circ\circ} \Delta_{\bullet\bullet\bullet\bullet\circ} \delta_{\bullet\circ\circ\circ\bullet} + x_{\bullet\circ\circ\circ\circ} \Delta_{\bullet\bullet\bullet\bullet\circ} \delta_{\circ\bullet\circ\circ\bullet} \tag{7.15}$$

The difference is easier to see with the symbolic expansion below.

```
fax = [[[2*x1,x2],[x2,0]],[[0,x1],[x1,2*x2]]]
fxa = [[[2*x1,0],[x2,x1]],[[x2,x1],[0,2*x2]]]
```

All the scalars are the same, but the positions in the array are different. This is due to differences in the broadcasting. The potential trouble comes when the gradient is being used. If the automatically calculated gradient is used in an expression it is important to make sure that the correct version of the gradient is used, otherwise the calculation might be incorrect.

7.6.3 Gradient with respect to an expression

When the gradient of an object a is calculated with respect to object b, the gradient calculation method will recursively calculate gradients of the dependencies of object a until the result is evaluated to either zero, which means that object a was not depending on object b in the first place, or the calculation eventually tries to calculate the gradient of b with respect to b. The gradient of b with respect to itself is an identity. This, however, will not be the case if object b is a copy of the original, and the gradient of the original is not stored inside, because the gradient then is zero. For variables and parameters the gradient with respect to the object itself is calculated when the object is created and stored inside the object, thereby avoiding the problem of calculating the gradient of the copy first. Doing so is always safe, because it will always make sense to find the gradient of an object with respect to either a parameter or a variable. On the other hand, the gradient with respect to an expression cannot in general be said to give a meaningful result. If the user needs

to calculate the gradient with respect to an expression, it is entirely possible, but the interpretation of the result is left to the user. No automatic gradient with respect to itself is therefore calculated. When the gradient with respect to an expression is calculated it is therefore important to make sure the gradient of this object with respect to itself is calculated first, like this:

```
a.grad(a)
b.grad(b)
```

This will install the correct gradient in all copies of this object. If this is not done, the calculated result might be erroneous.

The reason for this possible mistake is due to a difference between the internal and the external interpretation of objects. In the following expression two different copies,x_1 and x_2, of object x is used.

```
RGrad::expr(x[d, nil], a, x[nil, d]){
    |_x1, _a, _x2| _x1*_a*_x2
}
```

The gradient is the same for both copies of x, but the corresponding objects in the λ-function must be treated differently when the λ-function is differentiated in order to satisfy the chain rule. This means that the two copies of x must be distinguished, but at the same time give the same gradient. A gradient of an object is only calculated once, and a copy of this is used for all other needs. It is therefore important that the first calculation of the gradient is from the original object.

7.6.4 Misuse of goto statements

Goto statements are used to skip the execution of chunks of code. This can be a very practical feature, but it must be implemented and used with great care. When the graph is finalized it is ensured that each equivalent expression is calculated only once. It is also made sure that this happens in the code where it is first needed. When the entire code is executed it works without any unexpected side effects, but if some parts of the code are skipped it must be made sure that all the expressions that are needed for the chunk of code are up to date. This will only be a problem if parallel branches are implemented, or if a section of the code is skipped before it is calculated. The problem is best illustrated with a short example:

$$
\left.\begin{array}{l} T = T_\circ + \alpha\delta T \\ \text{if } T > 0 \text{ then} \ldots \end{array}\right\} \text{Chunk 1}
$$

$$
\textit{label}:
$$

$$
\text{if } T > 0 \text{ then} \ldots \} \text{Chunk 2}
$$

If chunk 1 is skipped the temperature is not calculated, and the test in chunk 2 will fail.

The intended use for the goto statement is to calculate only the first part of the code, typically only the first or maybe also the second order gradient, and then call the model again to do higher-order gradients, skipping the calculations that are already done.

Chapter 8

Code Generation

This chapter deals with output created from the RGrad language of Chapter 7. Once the graph is finalized, the objects in the graph can be exported to several different languages for different purposes. In general, the objects in the graph depends on each other, and it does not make sense to export individual objects. However, there is one exception: Expression and Halt objects represent entire sub-graphs, and it can therefore be of interest to export these objects as dot-files*. This gives a graphical representation of the complexity of the objects, which is useful for debugging. Table 8.1 summarises the different output capabilities implemented for the RGrad language.

Table 8.1: The different output possibilities implemented in the RGrad language (x = possible, o= not implemented).

	C	LaTeX	dot	matlab	ruby $>$ self	ruby $>$ binary
Graph	x	x	x	x	x	x
Expression	o	o	x	o	o	o
Halt	o	o	x	o	o	o

All output relies on a method which traverses the graph in the right order and translates the information stored in the objects to the desired output. The exportation facility is implemented as “plug-ins”. All plug-ins overload the same methods, and since the methods are overwritten dynamically only one exportation is available at a time. It is possible to swap between the different output methods by reloading the plug-ins. An example is shown below:

*dot is a language for writing graphs. There are several software tools available for visualising graphs written in this language

```
rgrad.finalize
rgrad.export('path') if RGrad::C::load?
rgrad.export('path') if RGrad::CRuby::load?
rgrad.export('path') if RGrad::CMex::load?
rgrad.export('path') if RGrad::Latex::load?
rgrad.export('path') if RGrad::Dot::load?
```

Listing 8.1: The method `finalize` will remove all duplicate objects from the graph, and prepare it for output. The `load?` methods overload one central method which generates the output, and if the overloading is successful, new output code is generated. The "path" must of course be replaced with the appropriate path for the output file.

8.1 C-Code

The main output from the implementation described in the previous chapters is ANSI C-code, which is divided into several different sections. First of all the exported code depends on a C implementation of the grammar described in Chapter 6. This library, called Pandora, is documented in Section 8.6. Next, each model generates four model specific functions: An initialization function allocating the memory needed, a computation function to calculate the results, a function returning a help string, and a function returning a time stamp. On top of this there is a general C-API which is the same for all the models. This API is used when defining the Ruby and the Matlab mex interfaces described in the following sections.

It is a goal to generate code which is both platform and compiler independent. To confirm the achievement of this goal, the code has been compiled on several different platforms, and with several different compilers. On Microsoft Windows, the Microsoft Visual C++ compiler, the Borland C++ compiler and the LCC compiler provided with Matlab have been used, and all worked satisfactory. The GCC compiler has been used on Linux, BSD and Mac OS X, all with success. This proves that the library and the generated code is standard which probably works with most C-compilers.

The code generation script creates one C source file and one C header file. All the constants used by the API-functions methods explained later are defined in the header file. In addition three model specific function prototypes are defined. These functions make up the model specific part of the C-API. In addition to the model specific files, there is a C source file, and a C header file implementing the Pandora library described in Section 8.6. These files and the C header file defining the C-API must be copied to the right location.

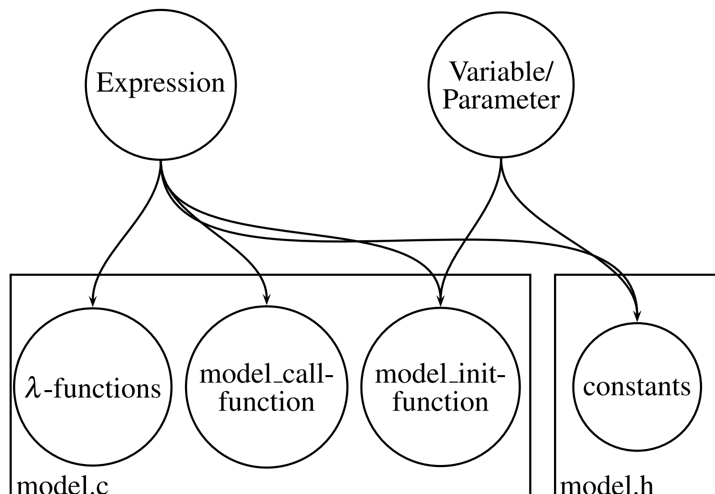


Figure 8.1: When RGrad objects are exported to C-code, code lines have to be written to several parts of the C-files. All objects that inherit from these objects will have similar behaviour.

One challenge when designing the code generation is that one object generates code blocks which must be written to several parts of the same file. A schematics of the work-flow for the different objects is outlined in Figure 8.1. The reason for this is the declarative nature of C. Variables need to be declared before they are used, thus each local variable generate two lines of C-code. Furthermore, since several functions are written to the same file, a way to assemble the C-file is needed. The solution is to create two small classes, one class for C-functions and another class for C-files. These classes define objects with queues for the different sections of a C program, and the exported code must simply be added to the correct queue. After all the model objects have been exported, those objects holding the C-code know exactly how to assemble the different parts of the C-file.

```
#include "rgradapi.h"
#define WILSON_DEFAULT_RETURN_FLAG 0
#define WILSON_MOLE_NUMBERS 1
#define WILSON_HALT_NEGN 2
#define WILSON_FIX_RGAS 3
#define WILSON_GOTO_2_DERIVATIVES 4
#define WILSON_G_NN_INV 17
#define WILSON_G_NN_PIVOTS 18
#define WILSON_UNIT_AMOUNT 20
void wilson_init(RGradapi *api, int sized0, int *arrd0);
char *wilson_timestamp();
```

```
char *wilson_help();
```

Listing 8.2: A typical header file generated by RGrad. All constants and function names are prefixed with the model name to avoid name clash.

8.1.1 Expressions

The expressions are the most complicated objects in the graph. These objects must create two different functions and write code to several places in the source file. Each expression also generates a separate function, which is the C-version of the λ-function, see Listing 8.3:

```
static double lambda12(double *);
static double lambda3(double *);

static double lambda12(double *a)
{
    return a[0] - a[1];
}
static double lambda3(double *a)
{
    return 8.3145 * a[0] * (-(log(a[1] / a[2])) + 1.0 - a[3]);
}
```

Listing 8.3: λ-functions are exported as single line functions. The prototypes are written to the beginning of the file, while the functions are written elsewhere in the file.

These functions are only meaningful in the context of the exported model, and are not intended for other use. They are therefore declared static. All functions have the same structure, and they take one argument, which is an array of doubles. As mentioned in Section 7.3, the names of the variables inside the λ-functions are given to reflect this, by dereferencing the array positions. The function body is written in one line, and a double is returned. In order to keep the number of exported lines at a minimum, all functions with duplicated function bodies are removed. This is done by creating a hash table of all the λ-functions while still in Ruby. The C string representation of the function body is used as a key to this table, and the function name is used as the corresponding value. A new entry is created if a particular function body is not found in this hash. This decouples the λ-function from the expression, and the hash table is used to give the name of the λ-function when the C-code for an expression is created. All the λ-functions are exported independently, based on the final hash table. This must of course be done after all the objects in the graph have been exported, otherwise some of the λ-functions can be undefined at the time they are referenced.

A similar approach is used for scientific units. Units are used for both consistency check when the objects are created, and to scale the calculations run-time, as mentioned in Section 7.1. The units are represented by an array of exponents in the C-code. Many expressions share the same units and instead of generating one array for each object, only unique exponent arrays are created which are reused several times. The unique unit coefficient arrays are written to the initialization function after all the objects in the graph have been exported.

```
void wilson_init(RGradapi * api, int sized0, int *arrd0)
{
    int u1[5] = { 1, 0, 0, 0, 0 };
    int u9[5] = { -2, 2, 1, 0, -2 };
    Dim *d0;

    d0 = makeDim(sized0, arrd0);
    storage->e[1] = allocPandora(5, u1, 0);
    storage->e[0] = allocPandora(5, u1, 1, d0);
    storage->e[15] = allocPandora(5, u9, 2, d0, d0);
    storage->z[14] = storage->e[15];
}
```

Listing 8.4: When memory for expressions are allocated the units and the dimensions must be known. Units are stored as an array of exponents. Dimensions are created based on information given into the function. The array *e* stores expressions, while the array *z* stores a pointer to the expression.

It is quite arbitrary which of the object copies that survive the graph finalisation and the removal of duplicate objects. If the dimensions of the surviving object are broadcasted, the broadcasting must be removed before the object can be exported. This can safely be done because no object is created with broadcasting in the first place, since the broadcasting makes sense only in the context where the object is used. The dimensions are needed to calculate the memory that must be allocated in order to store the result of the expression. This memory must be allocated before the calculations take place. It is not wise to allocate memory each time the code is executed, and the allocation is therefore done in the initialization function. One line of code is therefore written to the initialization function for each expression. See library function `allocPandora` on page 141 and `Storage` struct on page 139 for more details.

To calculate an expression one of the iterators described in Section 8.6 is used. Which one is determined in the actual object: If the expression is combined with a sum, the `siter` function is called. If none of the arguments of the expression are broadcasted, the `titer` function can be used. This is determined by checking the dimensions of all the arguments. If none of the above happens, the normal iterator (`niter`) is called. As is seen from the example below, all the functions have similar interfaces:

```
int wilson_call(void *ptr)
{
    int tt[2] = {1, 1};
    int tf[2] = {1, 0};

    siter(e[0], lambda1, 2, elem, tmp2, tmp1, tfarr, 2, dim, sfact, tf, d
        [0], d[0], p[1], tt, v[0], ft);
    sum(e[1], v[0], f, tmp1, sfact);
    niter(e[15], lambda11, 5, elem, tmp2, tmp1, tfarr, e[8], tf, e[9], tt,
         e[10], tf, e[11], ff, e[14], tt);
    titer(e[17], lambda12, 2, elem, tmp2, e[4], e[16]);
    matinv(e[15], e[19], e[20]);
}
```

Listing 8.5: The iterators `siter`, `titer` and `niter` are used much in the same way. See Section 8.6 for more details. The arrays *tt*, and *tf* are used for broadcasting. The names reflect the broadcasting, where *t* means true and *f* means false.

As explained in Section 7.2 on page 93, the user must add a label to those objects that should be available from the outside. This label is used to create a constant which is defined in the header file connected to the model. The value of the constant reflects the position in an array. This array is created in the initialization function, and a pointer to the corresponding object is placed in the position of that constant. The constant is used together with the API functions for retrieving and storing values, as explained in Section 8.2.

Two different sums were described in Section 7.2.3 on page 97. The sum that is being used for variables, parameters and previously calculated expressions is exported in the same way as the expression, except for the function call. The `sum` function on page 146 is used instead of the ordinary iterator, as can be seen from Listing 8.5.

8.1.2 Legendre transforms

The RGrad language is extended with Legendre transforms. Nothing new is required from the language except that an inverse is needed (see Chapter 2) when the Legendre transform is differentiated. This feature is, as mentioned earlier, not a part of the grammar described in Chapter 6, and the inverse is implemented solely to handle the Legendre transform, and nothing else. Even though the inverse is a purely internal construction, it can be practical for the user to access the result. The inverse of a matrix can be a source of problems if the matrix gets close to singular, and to aid the user in analyzing his results, the pivots from the inverse calculations are also returned. The inverse is an internal object, and is not available for any

manipulation in the RGrad language. It is therefore not possible to add labels to inverses, which means the inverse is not accessible from the outside. This is a bit inconvenient, and is fixed by assigning automatic labels to both the inverse and the pivots, provided that the original matrix has been given a label. The label of the matrix inverse will have the same root as the matrix being inverted, suffixed by "_inv" and "_pivots". The code written to the initialization function is of the same kind as for ordinary expressions, with memory allocated both for the inverse and the pivots. No λ-function is required for the inverse, but the line that calls the inverse function must be added to the computation function, see Section 8.6.15 on page 148 for more details. These hacks are unfortunate but necessary to handle the Legendre transform satisfactory.

8.1.3 Variables and parameters

Variables and parameters generate code for the initialization function only. As for the expressions, the broadcasting is ignored when the size of the object is determined. The memory needed to store the variables and the parameters, is allocated with the library function `makePandora` on page 140. The default values of the variables and parameters must be given to this function, and in order to do so these values are stored in temporary arrays, which are sent to the function. The default value of 1.0 is used if no values are given when the variable or the parameter is defined. The variables and parameters depend on units in the same way as the expressions, and any new combination of units generated by these objects is therefore added to the global list of units. Variables and parameters can also be accessed from the outside. This means that when a label is added, an extra entry is written to the initialization function and to the header file in order to make the content accessible.

```
double v0[3] = { 0.2, 0.3, 0.5 };
double p0[1] = { 8.3145 };
int D0[1] = { 3 };
int u1[5] = { 1, 0, 0, 0, 0 };
int u2[5] = { -1, 2, 1, -1, -2 };

storage->p[0] = makePandora(5, u2, 0, 1, NULL, p0);
storage->v[0] = makePandora(5, u1, 1, 3, D0, v0, d0);
storage->z[1] = storage->v[0];
storage->z[3] = storage->p[0];
```

Listing 8.6: Memory for variables and parameters are allocated very much in the same way as expressions, see Listing 8.4. `NULL` means that the parameter does not depend on dimensions and is thereby scalar.

In addition to variables and parameters, some special objects are exported in a similar way. Objects that are differentiated to zero can still be exported for completeness. These objects are exported as a parameter with all the values set to zero. Identities are also handled like parameters, but in order to deal with run-time changes of dimensions a special allocation function is needed for these objects, see 8.6.7 on page 142 for details. Constants are exported exactly as parameters, but they will never be used for calculations in the final code. They are added only to access the value from the outside. Changing the value of the constant will not affect the result of the computation because the value of the constant is incorporated directly into the λ-function where it is used.

8.1.4 Control statements

The control statements inherit most of their capabilities from either expressions or parameters, and the exported code is therefore the same but with some additional statements. Halt statements are exceptions to the rule, where a method to find the smallest value in a structure is used instead, see the `mins` functionon page 147 for more details. A halt statement is always reduced to a single value. The calculations stops if this value is less than or equal to zero. In order to recognize where the execution stops, the halt statements must be given a label. This label will correspond to a constant in the exported code. If the calculation stops, the negated value of this constant is returned.

```
  mins(h[0], lambda0, 1, elem, tmp2, tmp1, tfarr, 1, dim, d[0], v[0], t);
  if (h[0]->arr[0] <= 0)
      return -WILSON_HALT_NEGN;
  if (h[1]->arr[0] <= 0)
      goto flag_2_derivatives;
  siter(e[0], lambda1, 2, elem, tmp2, tmp1, tfarr, 2, dim, sfact, tf,
    d[0], d[0], p[1], tt, v[0], ft);
flag_2_derivatives:
  if (h[3]->arr[0] <= 0)
  return WILSON_FLAG_2_DERIVATIVES;
  niter(e[8], lambda5, 2, elem, tmp2, tmp1, tfarr, v[1], f, e[0], t);
```

Listing 8.7: The control statements will check whether the value is less than or equal to zero, and do the appropriate action.

Flag and goto statements are initialized like parameters, but again with some additional lines of code written to the computation function. A flag is implemented as a parameter that can be set from the outside. If this parameter is less than or equal to zero, the execution of the code stops at the point where the flag is set. An if-test in the code makes sure that the corresponding constant value is returned. This value can be used to figure out exactly where the execution stopped in the model

code. In opposite to the halt statement, the return value is always positive. The goto statements rely on these flags. A C goto label is written before each if-test. The goto statement has a similar if-test as the flag, but if the value is less than or equal to zero, the execution jumps to the flag specified rather than stopping the execution.

8.1.5 Additional output

The same graph which holds the objects to be exported is also responsible for generating the framework for the rest of the code. As mentioned earlier, the graph is responsible for defining four different functions. Three of these are accessible from the outside, while the computation function is an internal function which is accessible only through the API (see Section 8.2). As mentioned earlier, one C-file and one header file are always created. The graph object makes sure that the API header file is included in the model header file. This is required because the initialization function depends on this API. Furthermore, the prototype for the initialization function, the time stamp function and the help string function is written to the header file. The default return flag from the computation functions is defined as a constant, and is always zero.

The interface to the initialization function is model dependent. The reason for this is that the model dimensions can be changed runtime, and they must therefore be given as arguments to the function header. Since the number of dimensions in two different models will in general be different, the number of arguments to the initialization function will also vary. The first argument, however, is always the same. This is a pointer to the API struct defined in Section 8.2. For each of the dimensions two arguments are given: The first one is the number of components in the dimension array, and the second is an array of zeroes and ones to turn on and off the active components. A value of zero will remove the component from the calculations, while a value of one will keep it.

```
void wilson_init(RGradapi * api, int sized0, int *arrd0)
{
    Storage *storage;

    storage = mymalloc(sizeof(Storage));
    storage->ne = 22;
    storage->nv = 2;
    storage->e = mymalloc(sizeof(Pandora *) * 22);
    storage->v = mymalloc(sizeof(Pandora *) * 2);
    storage->elem = mymalloc(sizeof(Pandora *) * 5);
    storage->tfarr = mymalloc(sizeof(int) * 5 * 3);

    api->call = &wilson_call;
    api->get = &get;
```

```
    api->set = &set;
    api->add_to_values = &add_to_values;
    api->get_number_of_elements = &get_number_of_elements;
    api->get_rank = &get_rank;
    api->get_dim = &get_dim;
    api->free = &destruct;
    api->ptr = (void *) storage;
    api->last_constant = WILSON_UNIT_TIME;
    return;
}
```

Listing 8.8: The size of each array in the `Storage` struct is precomputed and hard-coded into the C-file and used when memory is allocated. All the function pointers needed in the API-struct are set in the initialization function.

The previous sections explain how the memory for the different parts of the model is allocated. All these (small) memory segments are stored in a `Storage` struct, as explained in Section 8.6.3 on page 139. This struct holds many arrays of pointers, and the size of each array. As explained in Chapter 7, the names assigned to the expressions, variables and so forth reflect the absolute position in these arrays. When all the objects have been exported it is possible to find out how large each array is, and allocate the required memory. The size of each array is also stored inside the `Storage` struct, and the dimension information given to the function header is used to create `Dim` structs (see Section 8.6.1 on page 138) which are also stored inside the `Storage` struct.

The API struct requires pointers to the different functions, and these pointers must be set before the initialization function returns. The library described in Section 8.6 is not needed directly by the user, but a pointer to the `Storage` is needed. Instead of forcing the user to include the header file from the Pandora library, the `Storage` struct is cast to a void pointer before it is placed inside the API struct. The user does not require to know what this pointer is, since it is only used by the other functions internally in the API.

```
char *wilson_timestamp()
{
    char *str = "2008-01-01T00:00:00Z";
    return str;
}

char *wilson_help()
{
    char *str =
    "This is an implementation of the Wilson model.\n
    The model is differentiated with respect to t and n.\n
    This is Legendre transformed with respect to n.";
    return str;
}
```

Listing 8.9: The help string and time stamp are exported in the same way as they

are given in the input language.

The last two functions defined by the graph take no arguments and simply return an array of characters. These functions only define the character arrays and return them. This could also have been done in the global workspace with the same effect to the end user. However, the use of functions gives a more flexible code, and function calls are more in line with the rest of the code. The help and timestamp functions should be regarded more like utility functions for the model than being a part of the API.

8.2 C-API

The C-API described in this section has matured from several design choices. The library described in Section 8.6 is not visible to the end user, therefore the API is created such that the inclusion of the library header file is not needed. It was also decided to create one struct that holds pointers to all the functions. This makes it easier for the end user to switch between models, since only the initialization function changes. The struct is populated by the initialization function. The prototype shown below, defines a total of eight functions, in addition to a void pointer and an integer. A Ruby interface and a Matlab mex interface based upon this C-API are described in the Sections 8.3 and 8.4.

```
typedef struct {
    void (*get)(void *ptr, const int cst, double *arr);
    void (*set)(void *ptr, const int cst, double *arr);
    void (*add_to_values)(void *ptr, const int cst, double *arr);
    void (*free)(void *ptr);
    int (*get_number_of_elements)(void *ptr, const int cst);
    int (*get_rank)(void *ptr, const int cst);
    int (*get_dim)(void *ptr, const int cst, int dim);
    int (*call)(void *ptr);
    void *ptr;
    int last_constant;
    int memsize;
} RGradapi;
```

Listing 8.10: The prototype for the common C-API

The void pointer is really a pointer to a `Storage` struct holding the allocated memory. Since the library is hidden for the end user, this pointer is cast to void and back again. The `last_constant` integer is set equal to the last constant which is defined in the model header file. This will coincide with the last position in the array which holds all the elements that can be accessed from the outside. The point of showing this constant to the API is to enable the end user to safeguard

the use of the various set and get functions. These are generic functions, and do not have any knowledge of the size of the arrays being used. The program will in fact crash if it for some reason is forced to access an array outside its bounds. It is up to the user of the API to make sure this does not happen. Finally, as a pure utility the integer `memsize` will tell how much memory is allocated for the model all together.

The `call` function requires only the void pointer in the struct. This function evaluates the computation function described earlier. The return value is an integer which tells the status of the calculation. A negative value means that one of the tests failed, a positive nonzero value means that the execution stopped at a user specified point, and a zero value is the normal exit.

The functions inside the API are, with the exception of the `call` function, generic functions which are independent of the model. Inside these functions no testing is done to prevent the user from doing stupid things. Therefore, three utility functions `get_number_of_elements`, `get_rank` and `get_dim` are implemented.

The `get_number_of_elements` function requires the void pointer and an integer to tell what array to retrieve the number of elements of. This integer must be one of the constants defined in the header file, where the name of the constant reflects what is returned. The `get_number_of_elements` function can be used to figure out how much memory is needed to store the result from the `get` function explained below, knowing the word length of course.

The `get_rank` function also requires the void pointer, and an integer. It returns the number of dimensions of the element that the integer points to.

The final utility function is the `get_dim` function. This function is used to find the size of a specific dimension. In the same way as with the other utility functions, this function needs the void pointer and an integer. In addition, an integer which tells what dimension the size is requested for must be provided.

The `get` function is used to copy numbers from the internal model storage out to the user. Typically, this will be calculated gradients or model parameters. The first two arguments to this function are the same as for the utility functions explained earlier. The internal data structure, defined in Section 8.6.2, is more complicated than the standard C data structure. Even though the numbers are stored in a standard C array it is not desirable to give away this pointer to the user, since the user then can destroy information needed elsewhere. The numbers are therefore copied. Another reason for copying the numbers is that the units used internally may be different from the external units. Therefore when the numbers are copied out they are at the same time converted from internal units to external units. The last ar-

gument to the get function is therefore an array which will hold all the numbers provided. If the array is too small, the program will crash, and to make sure the array is big enough, the get_number_of_elements function should be used. Regardless of the structure of the stored element, the numbers are stored in a flat array with row-major ordering.

The set function is the opposite of the get function and is used to copy values from the user application into the system. The arguments are the same as for the get function, but in this case there is an array given as the last argument which contains the numbers to be stored. Regardless of the structure of the stored array, the numbers are given in a flat array with row-major ordering. The numbers are converted from external units to internal units as they are copied. The array given must be of the same size as the element that is stored, otherwise the program will crash. The get_number_of_elements function should be used to safeguard for this before the set function is called.

The add_to_values function is a special version of the set function. Instead of copying directly into memory, the numbers given are added to the values residing in memory (after units conversion of course).

The final function in the API is the free function. This is used to release the memory that is allocated by the initialization function. The only argument needed is the void pointer.

8.3 Ruby Interface

Section 8.2 describes the API which has been used to create an interface to the model from the interactive scripting language Ruby. Ruby has a C-API which makes it relatively easy to extend the language with third party code. The code is compiled, and linked into a library that can be used alongside with native Ruby code. How to extend Ruby is well covered by Thomas et al. (2005), and those details are not repeated here. The Ruby interface described in this section wraps all the functionality into one class. An instance of this class will then be a self contained model object.

```
a = Wilson::new
=> #<Wilson:0xb796b9fc @storage=#<Pandoraswrap:0xb796b9d4>, @memsize=3188>
b = Wilson::new([1,0,1])
=> #<Wilson:0xb7968aa4 @storage=#<Pandoraswrap:0xb7968a7c>, @memsize=2748>
```

Listing 8.11: Two different instances of the Wilson model are created. Notice that the amount of memory is reduced when one component is removed from the model.

In this section the definition of the Ruby interface and the use of it is discussed. Most of the interface is hand coded and eventually copied for each model that is exported. The only model dependent code that needs to be generated, is the list of constants used in the C-API. This is done by collecting all the labels in the graph defining C constants to set the value in the C-code.

The Ruby interface defines a new class, and objects of this class will be separate instances of the model. When an object is created, the initialization function described in Section 8.1 is called. This will allocate memory for computing the model. A pointer to this memory is stored inside a Ruby object as an instance variable. Even though it is stored inside the Ruby object, the memory can only be accessed from C. For convenience, two different ways of creating new objects are implemented: If no arguments are given to the constructor it is tacitly assumed that the default dimensions of the model are accepted. This is possible because C arrays representing the default dimensions are defined in the C-function that initialize the objects. These arrays are given to the model initialization function. The default dimensions can also be overridden by supplying arguments to the constructor. The number of arguments required is given by the total number of different dimensions used by the model. For each dimension that is used, an array of zeroes and ones must be supplied. A value of zero means that the component in that position is removed from the model, and a value equal to one means that the component is included. The number of elements in the array must be at least the same as the size of the corresponding dimension, or else an error is raised. If the array contains more elements than is defined in the RGrad language, an unknown component is added. This situation is legible, but all the parameters will be initialized to 0.0. This means that the entire parameter space (all values) must be provided by the user.

The C memory allocated for the object is stored in the instance variable `memsize`. This is accessible from Ruby and will tell how many bytes of memory the model object requires.

```
Wilson::API.constants
=> ["PAR_LIJ", "Z_N", "HALT_NEGN", "FLAG_1_DERIVATIVES", "Z", "
    MOLE_NUMBERS", "UNIT_MASS", "G","G_N" ...]
```

Listing 8.12: The model constants are stored inside the submodule API

It is clear that the constants defined in the model header file described in Section 8.1 must somehow be available from Ruby. Several different approaches were considered, but it was finally decided to create a submodule of the model class, and transfer the values of the C constants to equivalent Ruby module constants in that module. For all models this module is simply called `API`. These constants can be used directly, but it is also possible to use a string version of the constants name

in the method calls. In contrast to the constants themselves, the strings are case insensitive because they are converted to upper case letters runtime. The string is concatenated with the model class name and the API module to reference to the Ruby constant in the background. If the given string given is not referring to a valid constant an error is raised, and if the constant for some reason is outside the range of the cumulative list of constants in the model, an error is also raised.

```
a.call => 0
b.call => 0
a['g_n']
=> [-2352.80210798469, -2000.51492195302, -1096.25793707823]
b['g_n']
=> [-2766.68060502227, -396.166780411]
```

Listing 8.13: A return value of 0 is the default value, meaning that all expressions in the model were calculated, and the results can then be retrieved.

The `call` method is used to start the model calculations. This method requires no arguments, and returns the integer value from the C-code.

In Ruby it is possible to retrieve values out of the object by using the `[]` operator. This method takes one mandatory argument which is a Ruby constant, or its string equivalent, and returns a Ruby array containing the corresponding values. If the returned object is a scalar it will be converted to float. This is more or less the same that is required for the `get` function in the C-API, with the exception of the pointer. The pointer is held by the object, and is automatically included in the function call in the background.

```
a['temperature'] = 300
=> 300
b['mole_numbers']
=> [0.2, 0.5]
b.add_to_values('mole_numbers', [0.1, -0.1])
b['mole_numbers']
=> [0.3, 0.4]
```

Listing 8.14: Setting and retrieving values through the Ruby interface.

In the same way, values can be set with the `[]=` operator. Here the right hand side argument must be an array with the correct size, but in the case where the element is a scalar, it is sufficient to supply a float. No operator is defined for the `add_to_values function`, but a method with the same name is defined. The added values are given as an argument to the method together with the constant identifying what storage element is changed.

In addition to the ordinary `set`, `get` and `add_to_values` functions, there are three special functions: `get_by_ptr`, `set_by_ptr` and `add_by_ptr`. These functions are practical when Ruby is used on top of another C-program. Instead

of sending and receiving Ruby arrays to and from the functions, a memory pointer is given instead. The first argument tells which element to set or get, the second argument is the memory pointer to the first element in the C array held by the other external caller. The third argument tells the total size of the memory segment which is operated upon. This is used to make sure the array has the correct size, and if not an error is raised. The last argument tells how much the pointer should be moved for each increment. This enables a different memory layout than the one used in the library described here. The external C array could for instance store imaginary values in every second position of the array.

Two other utility functions have also been implemented: The `to_a` method is used to list all numbers available inside the model. These are returned as an array of arrays, where the arrays are sorted in the alphabetical order of the constants. The `to_a_with_str` works the same way as `to_a`, but a string representation of the constant name has been added as well. Additional utility functions can easily be added, as they are added as plain Ruby code.

```
Wilson::HELP
=> "This is an implementation of the Wilson model.
The model is differentiated with respect to t and n.
This is Legendre transformed with respect to n."

Wilson::TIMESTAMP
=> "2008-01-01T00:00:00Z"
```

Listing 8.15: The help string and time stamp are stored as constants in Ruby.

Finally the help string and time stamp functions described in the previous section are available as class constants, but the `free` function described in the API is not mentioned here. This is because the Ruby garbage collector takes care of deleting unreferenced objects. A pointer to the `free` function is given to Ruby when the objects are created so the garbage collector knows how to release the memory that is allocated.

8.4 Matlab mex Interface

Section 8.2 describes the API which is used to create an interface to the model from Matlab. The Matlab mex API defined by MathWorks (2008) makes it possible to use C-functions from Matlab. The function prototype of the C-function which is called from Matlab is predefined. This means that one C-file must be generated for each Matlab function that is needed.

A total of six functions are defined through the Matlab interface, but only one is model dependent. This is the initialization function which is given a name that is

related to the model name. This means that it is relatively easy to use different models within the same Matlab script, because only the lines where the model is initialized need to be changed*.

```
>> a = wilson_init
a =
   137760008
>> b = wilson_init([1,0,1])
b =
   137369416
```

Listing 8.16: Two different instances of the Wilson model are created. The memory pointer returned must be kept and used by other functions.

To avoid several model specific functions, the behavior of the initialization function depends on the argument given. The default behaviour with no arguments is to initialize the model with the default dimensions. This call returns a memory address, which is the pointer to the allocated memory. The memory address must be stored in the Matlab workspace, otherwise an error is raised. The address must not be altered by the user as this will make the function crash. This way of handling the memory pointer is relatively fragile, but no other way of dealing with the problem was found. The default dimensions can be overridden in the same way as in the Ruby interface, i.e. by switching components on and off using 0 or 1 in the array argument.

```
>> s = wilson_init('struct')
s =
    flag_1_derivatives: 9
                     g: 8
                   g_n: 7
             halt_negn: 2
          mole_numbers: 1
               par_lij: 6
             unit_mass: 22
                     z: 15
                   z_n: 16
                   . . .

>> wilson_init('help')
ans =
This is an implementation of the Wilson model.
The model is differentiated with respect to t and n.
This is Legendre transformed with respect to n.
>> wilson_init('timestamp')
ans =
2008-01-01T00:00:00Z
```

Listing 8.17: The initialization function responds appropriately to different arguments. The struct will be used for setting and retrieving values from the model.

---

*This requires that the variables, parameters and results used are given the same names in all models.

The C-constants give access to variables, parameters etc., but they cannot be directly transferred to Matlab, and two different ways of accessing these constants are implemented. Both are accessible through the initialization function, either as a struct or as a cell array. The different data structures are obtained by specifying a string named “struct” or “cell” to the initialization function. The field names of the returned struct reflect the constant names, and the values are the same as the constant values. The cell array contains a cell array in each cell. Each cell array contains a string representing the constant name, and the number of the constant. The help string is available by giving the string “help” to the initialization function, while the time stamp is asked for by using the string “timestamp”.

The first argument to the rest of the functions described in this section is the memory address provided by the initialization function. The `rgrad_get` function requires in addition to the memory address a constant which tells what element to retrieve. One-dimensional elements are returned as column vectors, while two dimensional elements are returned as ordinary Matlab matrices. However, Matlab stores matrices with column major ordering, while the C-code described in this chapter stores values with row major ordering. A matrix might therefore be the transposed of what is expected. If the requested element has more than two dimensions, the numbers are returned as a column vector, without structure and with row major ordering.

The `rgrad_set` function is used to set the element of the second argument. The last argument must be a matrix with the correct number of values, otherwise an error is raised. There is no problem giving a multidimensional array to the function, but it must be kept in mind that Matlab stores values with column major ordering, whilst the C-library stores values with row major ordering. Matrices must therefore be given as transposed entities of what is natural in Matlab. The `rgrad_add_to_values` function works in the same way as the `rgrad_set` function, but the values are added to the ones already in memory.

```
>> rgrad_call(a)
ans =
     0
>> rgrad_call(b)
ans =
     0
>> rgrad_get(a, s.g_n)
ans =
   1.0e+03 *
   -2.3528
   -2.0005
   -1.0963
>> rgrad_get(b, s.g_n)
ans =
   1.0e+03 *
   -2.7667
```

```
   -0.3962
>> rgrad_set(a, s.temperature, 300)
ans =
   300
>> rgrad_get(b, s.mole_numbers)
ans =
    0.2000
    0.5000
>> rgrad_add_to_values(b, s.mole_numbers, [0.1, -0.1])
ans =
    0.1000
   -0.1000
>> rgrad_get(b, s.mole_numbers)
ans =
    0.3000
    0.4000
```

Listing 8.18: A return value of 0 is the default value, meaning that all expressions in the model were calculated.

The function `rgrad_call` runs the model and returns an integer from the C-code to Matlab. When the user has finished the calculations, the allocated memory should be released with the `rgrad_free` function. Once this function is called, the memory pointer becomes useless. An attempt to use this after the memory is freed will make Matlab crash.

8.5 Additional Features

This section deals with output which has attained less focus or has been made more out of curiosity than usefulness. This means that some of the output codes worked during the early versions of the program, but have not been updated to handle the last-minute features.

LaTeXcode

LaTeX is the natural choice of writing mathematical texts, not so much for writing computer code but it is still useful for reporting automatically generated material. An example output is shown in Figure 8.2. In addition to documenting the exported objects, an estimate of the number of numerical operations needed to calculate the result is reported.

This is possible by manually counting the operations needed in each iterator functions. A distinction is made between the operations needed to calculate the positions in the arrays of the iterators, and the operations needed in the λ-functions,

see Section 8.6 for more details on iterators. Each operator has its own weight, which is currently set to 1 for all operators. However, this is just an estimate of the complexity of each expression, not an exact measure. Memory access is for instance assumed free in this case, but comparing different expressions and models should to some extent be possible. The accumulated number of expressions is also calculated. An excerpt of the compiled LaTeX output of the Wilson model is shown below.

v[0]^3
p[0] = 8.3145
p[1]$^{3\times 3}$ = $[1,2,3,3,1,2,2,3,1]$
Total number of Pandora internal operations = 2678
Total number of λ binary operations = 222

λ	Name	Alias	Size	$\sum$	lop	clop	op	cop
λ_1	h[0]	haltnegn	3		0	0	7	7
λ_5	e[3]	gn	3		21	30	28	507
λ_{11}	e[9]		3,3	$\circ\bullet\circ$	9	74	627	1306
λ_{21}	e[21]	znn	3,3		9	222	19	2678

$\lambda_1 = v[0]_\bullet$
$\lambda_5 = 8.3145 * v[1]_\circ * (-\log(e[0]_\bullet/e[1]_\circ) + 1 - e[2]_\bullet)$
$\lambda_{11} = i[1]_{\circ\bullet\bullet} * p[1]_{\bullet\bullet\circ}$
$\lambda_{21} = -e[19]_{\bullet\bullet}$

Figure 8.2: All parameters are documented by showing their default size and value. For expressions, the λ-function is shown, the size of the expression, and a prospective sum is shown. If any alias is given, this is also given. In addition, an estimate of the number of operations needed to calculate the result is made. lop is the number of λ-function operations needed for this particular expression. op is the number of operations needed to calculated array positions internally in the iterators. clop and cop are the corresponding cumulative values.

Dot files

Models and expressions can be visualised by means of the dot language, which is a textual language for representing graphs. In this case the graph representing an expression or a complete model is generated by traversal of the collected objects. This graph is only meant to be a part of the model documentation. The graph gets

quickly complicated, but it is nevertheless of interest to have an understanding of the complexity of the model or expressions. Figure 8.3 shows a small example from the Wilson model described in Chapter 7.

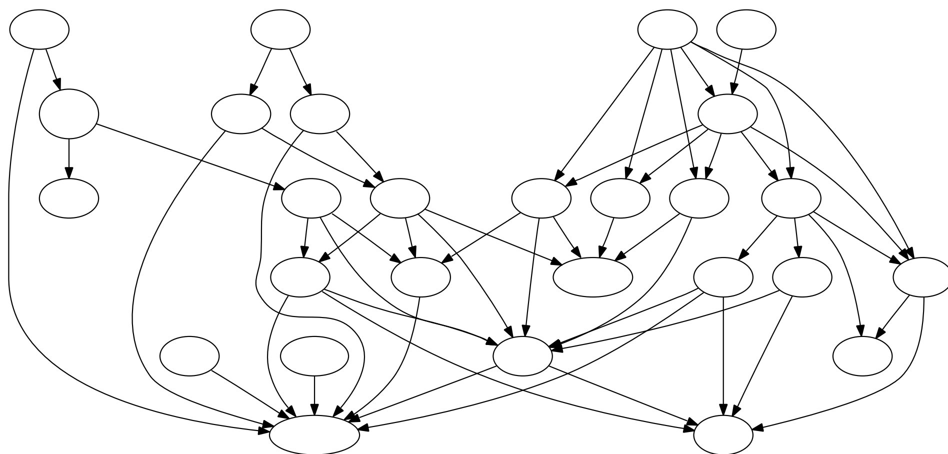


Figure 8.3: A graph from dot file generated from the Wilson model. The important thing is to show the complexity of the model, therefore the names of the nodes have been omitted. In general the root nodes of the graph are located at the top, and the leaf nodes are located at the bottom.

Binary operations

For some applications speed is more important than flexibility of the code. It is therefore possible to export the code as a set of binary operations. This gives shorter computation time as all the overhead is removed. The problem is that the flexibility is removed from the final code as well. For instance only the variables are changeable. The binary operations are achieved in two steps by using Ruby. First of all, the library described in Section 8.6 must be implemented in Ruby. The exported code will then use this library to generate the binary operations. This library uses operator overloading to collect the binary expressions. The same code that holds the internal representation of the λ-functions is used. This means that external programs, such as Maple, can be used to simplify the binary expressions before they are written to file.

Ruby to Ruby

This feature was implemented only to prove that the RGrad language is closed. The idea is that it should be possible to regenerate the input file automatically. This is taken as proof that the grammar defined in Chapter 6 is a closed. The model can be differentiated to arbitrary order, and the script will generate a valid input file which can be read and exported like any other input file.

Matlab

The exported Matlab script requires that the library which is described in Section 8.6 has been implemented in Matlab. The exported code is valid Matlab code, but does still depend on this library. The exportation worked fine, but the performance of the code was terrible, and further development of the feature was abandoned. The mex interface is actually a better alternative, see Section 8.4 on page 132 for more details.

8.6 Library Implementation

This section covers the documentation of a C library which all the exported C-code relies on. This library is called Pandora, and implements the iterators needed for calculating the results in the language described in Chapter 6. The library also defines functions for allocating memory segments needed for storing each element of a model. The numbers are stored in different structs, which are also described in this section. The library only depends on standard C libraries. A short description of all the implemented functions is given below. The function prototype for each function is also shown.

8.6.1 `Dim` struct

```
typedef struct {
    int rank;   /* The number of elements in dims; */
    int *dims;  /* An array of 0 and 1, 1 means the component is active */
} Dim;
```

Listing 8.19: Prototype of the `Dim` struct

This struct is used to store information about the dimensions used within the Pandora library. The `Dim` struct has two members and it has two distinct uses, as will

be explained later. The dims member is an array of 0 and 1, where 1 means that the corresponding component is active, while 0 means that the corresponding component is inactive. The rank member tells the total number of active components in the `Dim` struct. Examples of use of `Dim` structs are shown in Section 8.11 on page 129 and in Listing 7.2.1.

8.6.2 `Pandora` struct

```
typedef struct {
    double *arr;    /* Data array containing all the numbers */
    int *dim;       /* Array containing the dimensions of the struct */
    int *units;     /* Array with exponents of units */
    int rank;       /* The rank of the struct */
    int nel;        /* The total number of elements in arr */
} Pandora;
```

Listing 8.20: Prototype of the `Pandora` struct

This struct is used to store numerical values given in the initialization function or calculated from the iterators. The struct has five members in total: One array holding the numerical values, one array holding the dimensions, one array holding the exponents of the units, and two members holding the rank and the number of numerical values in total respectively.

8.6.3 `Storage` struct

```
typedef struct {
    int *tmp1,*tfarr;/* Temporary memory segments used frequently */
    int *dim, *sfact;/* Temporary memory segments used frequently */
    double *tmp2;    /* Temporary memory segment used frequently */
    Pandora **elem;  /* Temporary memory segment used frequently. */
    int nv;          /* The number of variables */
    int np;          /* The number of parameters */
    int ne;          /* The number of expressions */
    int ni;          /* The number of identities */
    int nh;          /* The number of halt/flag statements */
    int nd;          /* The number of dimensions */
    int nc;
    Pandora **v;     /* Array with all the variables */
    Pandora **p;     /* Array with all the parameters */
    Pandora **e;     /* Array with all the expressions */
    Pandora **i;     /* Array with all the identities */
    Pandora **h;     /* Array with all the halt/flag statements */
    Pandora **c;
    Pandora **z;
    Dim **d;         /* Array with all the dimensions */
} Storage;
```

Listing 8.21: Prototype of the `Storage` struct

This struct is used to keep the heap memory allocated for one model. The struct has 21 members in total. Six of these members are used for temporary memory

segments used by the iterator functions. The `Pandora` structs used to describe the model are stored inside the `Storage` struct in arrays. In order to make the code more readable, `Pandora` structs representing different types are stored in separate arrays. The sizes of these arrays need to be stored inside the `Storage` struct too. In total seven different arrays are used, and an array of all the `Pandora` structs that could be reached from the outside is created.

8.6.4 `makeDim` function

```
struct dim *makeDim(int size, int *dims)
```

This function allocates memory for a `Dim` struct, where the first argument tells the size of the second argument. As described in Section 8.6.1, the `Dim` struct has two members. The array of active components is the same as the array given to the `makeDim` function. The number of active components must be determined by counting the components that are turned on. The required information is stored inside the allocated struct, and the memory pointer is returned.

8.6.5 `makePandora` function

```
Pandora * makePandora(int nu, int *units,
    int rank, int nel, int *dim, double *arr, ...)
```

This function allocates memory to the `Pandora` struct that is initialized with user supplied values, see Section 8.6.2 on the preceding page for details about the `Pandora` struct. Since it is possible to both increase and decrease the component lists of the model runtime the function gets more complicated. The function takes six mandatory arguments in addition to a variable argument list. The first two arguments are related to the units. The first argument tells how many units there are inside the model, while the second argument is an array holding the exponents of the units. The next argument gives the rank of the array constructed. The fourth argument gives the number of elements in the array if the dimensions remain unchanged. Then an array of dimensions is given, and finally an array with the values. All the `Dim` structs needed for creating the `Pandora` struct is given to the variable argument list.

The memory for the `Pandora` struct is allocated first. The next task is to allocate memory for holding the dimensions of the `Pandora` struct. The rank of each `Dim` struct given is used to set the appropriate dimensions in the `Pandora` struct. The dimensions used in the `Pandora` struct can be changed runtime. This will obviously affect the amount of memory needed for the struct, and the number of

active components of each dimension. The rank member of the `Dim` struct gives the number of active components for each dimension. For further details on `Dim` struct, see Section 8.6.1 on page 138. The total number of elements stored inside the `Pandora` struct is the product of the number of active components and the dimensions.

An array of doubles with the correct size is allocated once the total number of elements of the `Pandora` struct has been determined. The `Pandora` struct is initialized with the values given to the function if the size of each dimension is equal to or smaller than the default dimensions. Otherwise all the elements are set to zero. Since the number of components can be reduced runtime some of the values given might not be used. In order to determine which values to store inside the `Pandora` struct, and which to ignore, the `Dim` structs are used. Each `Dim` struct has an array of 0 and 1 which tells whether the corresponding component is active or not. An inactive component means that the corresponding value should not be stored inside the `Pandora` struct. Finally the required members in the `Pandora` struct are set, and the struct is returned.

8.6.6 `allocPandora` function

```
Pandora * allocPandora(int nu, int *units,
    int rank, ...)
```

This function allocates memory for the `Pandora` struct that is used to store results from the computation function. The function takes three mandatory arguments and a variable argument list. The first argument gives the number of units in the model. The next argument is an array of exponents of the units. The last mandatory argument is the rank of the `Pandora` struct which is created. The variable argument list contains `Dim` structs. This is because it is possible to reduce the dimension spaces runtime, and the `Dim` structs are used to determine the actual number of elements in the `Pandora` struct.

The memory for the `Pandora` struct, the array for the unit exponents, and the dimensions array are allocated first. Next the dimensions and the total number of elements is determined by retrieving the `Dim` structs from the variable argument list. The size of each dimension is given by the rank member in the struct. The total number of elements is determined by calculating the product of the number of active components in all the dimensions. When this size is known it is possible to allocate memory for the array of doubles holding the elements in the `Pandora` struct. The memory is allocated such that all the elements are set to zero initially.

Finally, all the members of the `Pandora` struct are set and the struct is returned, see Section 8.6.2 for a description of the `Pandora` struct.

8.6.7 `makeeyePandora` function

```
Panodra * makeeyePandora(int nu, int *units,
    int rank, ...)
```

This function allocates memory for the `Pandora` struct that is initialized as an identity. The function takes three mandatory arguments, and a variable argument list. The first argument gives the number of different units in the model. The next argument is an array with exponents of the units. The last argument is the rank of the `Pandora` struct. The variable argument list contains `Dim` structs, which are used to calculate the actual dimension and the actual number of elements in the `Pandora` struct.

The memory for the `Pandora` struct and two arrays for the dimensions and the unit exponents are created first. Next the size of each dimension is copied to the `Pandora` struct, and the total number of elements is calculated. After that the unit exponents are copied into place, and an array holding all the numerical values is allocated, and initialized to zero. Finally, the diagonal or quadragonal and so forth is set to one. The members of the `Pandora` struct are set and the struct is returned.

8.6.8 `niter` function

```
void niter(Pandora *pan, double (*fun)(int, double *),
    int nl, Pandora **elem, int *tmp1, double *tmp2,
    int *tfarr,  ...)
```

The function takes seven mandatory arguments and a variable argument list. There is at least one additional argument. The first argument to the function is the `Pandora` struct which holds the memory of the calculations. The next argument is a pointer to the λ-function. The number of variables to be operated upon is given as the next argument. This is instead of using a stop element in the variable argument list. The next four arguments are pointers to temporary work arrays that are allocated during the initialization phase to store numbers temporarily within the function. See Section 8.6.3 on page 139 for further information. The variable argument list contains pairs of `Pandora` structs and broadcasting arrays.

After declaring temporary variables, the function unpacks the variable argument list. The variable argument list will be twice as long as the number of variables required inside the function because of the broadcasting which is connected to each `Pandora` struct. Each `Pandora` struct given to the function is stored inside a temporary array given to the function. The broadcasting is also stored inside a temporary array. The broadcasting is itself an array and the work array should logically be an array of arrays, but it was chosen to store it in a flat array and use a macro function to convert back and forth. This means that all the elements of the broadcasting array needs to be copied into the new array. This might seem like extra work but this is required because the values are overwritten, and the broadcasting arrays are used several times the values need to be copied anyway.

All the elements in the `Pandora` struct are stored in a consecutive C array regardless of the rank of the element. Since the broadcasting repeats some of the dimensions, the array is not necessarily read in consecutive order. When the array is used, the position must be calculated each time based on the dimensions of the `Pandora` struct, its corresponding broadcasting and the position in the array with no broadcasting. The following example will show how this is done.

First the dimensions of the `Pandora` struct and its corresponding broadcasting is used to calculate a factor array which will be used later on in the next paragraph. In the array of dimensions attached to the `Pandora` struct, the dimension to the right is the fastest running dimension. The factor array will have the same number of elements as there are dimensions in the resulting `Pandora` struct. The factors calculated starts at the fastest running dimension, and in the position where the first active dimension is located, the factor will be 1. This corresponds to incrementing the pointer in the array of values with 1 each time. The next factor is calculated by multiplying the previous factor with the size of the corresponding dimension. If the dimension is broadcasted, the corresponding factor will be 0. To show how this works, look at the four dimensional array with dimensions [5,4,6,7] when the broadcasting is taken to be [1,1,1,1], meaning no broadcasting in any of the dimensions. The factor array is then [4*6*7, 6*7, 7*1, 1]. If, on the other hand, some of the elements are broadcasted, the factor array will be different. For instance if the broadcasting is [1,0,1,0], the factor array becomes [6,0,1,0]. The zero means that the counter for that dimension has no effect on which element is picked from the array. This is valid for all `Pandora` structs given to the function.

The main loop of the `niter` function iterates over all the elements of the resulting `Pandora` struct. For each element a few things need to be done. As mentioned in Section 8.6.2, all elements are stored in a consecutive C array. The position in each virtual sub-array is calculated in the following manner: The flat array is the starting point and the position at the fastest running dimension is calculated first.

This calculation requires the modulo operator which returns the remainder of the division between the position in the flat array, and the size of the dimension. This operation is repeated for the next dimension, but now the position is reduced by calculating the floor of the position divided by the size of the current dimension. This procedure is repeated for for all the dimensions, and gives a counter which in combination with the factor array from the previous paragraph is used to calculate the correct position in each `Pandora` struct given to the function. For each element in the resulting `Pandora` struct, one element from each argument to the function must be collected. This is then given to the λ-function and the result is stored into the result struct. The position is found by adding the multiple of the factor array and the counter for each dimension. However, to unravel the mystery of broadcasting, factor arrays and dimensions, an example is probably needed:

Assume that the resulting array has the dimensions [3,2,2], and the argument array has the dimensions [3,2] with the broadcasting [1,0,1]. The factor array is calculated first. This yields [2,0,1]. Finding the position of the first element is of course trivial because this is always equal to zero. The rest is show in Table 8.3

Table 8.3: An array with dimensions [3,2] broadcasted to dimensions [3,2,2] will consecutively use the following position to retrieve values.

dim3	dim2	dim1	position
mod(0,2) = 0	mod(0,2) = 0	mod(0,3) = 0	1*0 + 0*0 + 2*0 = 0
mod(1,2) = 1	mod(0,2) = 0	mod(0,3) = 0	1*1 + 0*0 + 2*0 = 1
mod(2,2) = 0	mod(1,2) = 1	mod(0,3) = 0	1*0 + 0*1 + 2*0 = 0
mod(3,2) = 1	mod(1,2) = 1	mod(0,3) = 0	1*1 + 0*1 + 2*0 = 1
mod(4,2) = 0	mod(2,2) = 0	mod(1,3) = 1	1*0 + 0*0 + 2*1 = 2
mod(5,2) = 1	mod(2,2) = 0	mod(1,3) = 1	1*1 + 0*0 + 2*1 = 3
mod(6,2) = 0	mod(3,2) = 1	mod(1,3) = 1	1*0 + 0*1 + 2*1 = 2
mod(7,2) = 1	mod(3,2) = 1	mod(1,3) = 1	1*1 + 0*1 + 2*1 = 3
mod(8,2) = 0	mod(4,2) = 0	mod(2,3) = 2	1*0 + 0*0 + 2*2 = 4
mod(9,2) = 1	mod(4,2) = 0	mod(2,3) = 2	1*1 + 0*0 + 2*2 = 5
mod(10,2) = 0	mod(5,2) = 1	mod(2,3) = 2	1*0 + 0*1 + 2*2 = 4
mod(11,2) = 1	mod(5,2) = 1	mod(2,3) = 2	1*1 + 0*1 + 2*2 = 5

8.6.9 `titer` function

```
void titer(Pandora *pan, double(*fun)(int, double *),
    int nl, Pandora **elem, double *tmp2, ...)
```

This function is a special version of the `niter` function discussed in Section 8.6.8. It assumes that there is no broadcasting, and is thereby much simpler than the `niter` function.

The `titer` function takes five mandatory arguments in addition to a variable argument list. The first argument is a `Pandora` struct which stores the results from the function. The next argument is the λ-function. The number of elements in the variable argument list is given as the third argument. The final two arguments are memory segments used temporarily inside the function.

First, all the `Pandora` structs given to the variable argument list are collected. The main loop is a loop over all the elements in the result struct. Since there is no broadcasting here, it is trivial to find the position in the different arrays. This will be the same for all the arrays. For each element, one number from each argument struct is collected, and sent to the λ-function. The results are stored in a `Pandora` struct.

8.6.10 `siter` function

```
void siter(Pandora *pan, double (*fun)(int, double *),
    int rank, int *stf, int nl, Pandora **elem,
    int *tmp1, double *tmp2, int *tfarr, int *dim,
    int *sfact, ...)
```

This function is a combination of the `niter` function and the `sum` function. It is implemented in order to save memory in connection with inner products which are of the same nature as matrix multiplication in linear algebra. Instead of expanding a matrix product to three dimensions and then do the summation, the summation is done during the main iteration, as it is the case in all linear algebra packages. The code given here also works for arrays with three or more dimensions, and the sum can be calculated over more than one dimension at the same time.

The niter function takes eleven mandatory arguments, in addition to a list of variable arguments. The first argument is a `Pandora` struct which holds the result. The next argument is the λ-function to be used. In the `niter` function, the rank of the problem was given in the resulting `Pandora` struct. In this case there is a rank reduction and the rank of the unreduced system must therefore be given as an argument to the function. The next argument is an array that tells which dimensions to sum and which to keep. In order to avoid using a stop element in the variable argument list, the number of `Pandora` structs given in this list is given as the next argument. The next six arguments are memory segments used temporar-

ily inside the function. Instead of allocating the memory each time the function is called, it is allocated once and reused in all function calls, as for the `niter` function.

The number of elements to be summed is unknown since the size of the dimensions can be changed during the initialization. The first arguments in the variable argument list are therefore the appropriate `Dim` structs. The rest of the arguments are pairs of `Pandora` structs and broadcasting arrays.

First, the dimensions and number of elements of the overall problem* must be determined. This is done by calculating the number of active components in each `Dim` struct. The total number of elements is the product of all the numbers of active components.

Next the `Pandora` structs and the pertaining broadcasting is collected from the variable argument array, and stored inside their work arrays. After that, the factors for calculating the positions inside the arrays are computed, see Section 8.6.8 for details. An equivalent array needs to be computed for finding the position in the resulting array. The procedure for calculating the factors are equivalent to the ordinary iterators.

The main loop iterates over all the elements which had been present in the resulting array if it had not been summed. In any other respect the main loop is equivalent to the one explained in Section 8.6.8, except for the way the result of the λ-function is stored. In this case the function result is added to the value which is already present in the array. This requires that all the array elements are initialized to zero before the calculation starts (done automatically).

8.6.11 `sum` function

```
void sum(Pandora *result, Pandora *from,
    int *stf, int *tmp1, int *sfact)
```

This function is used to sum `Pandora` structs. The function takes five arguments. The first argument is a `Pandora` struct in which the result shall be placed. The next argument is the `Pandora` struct to be summed. As the next argument to the function, an array with 0 or 1 is given to the function. A value of 0 means that the dimension is removed and a value of 1 means that the dimension is kept. The last two arguments are memory segments used temporarily in the function.

*The dimensions and number of elements of the result if no sum had been called

Factors to calculate the position in the final array is computed first, see Section 8.6.8 for details. The factors are stored in one of the work arrays given as input to the function.

The main loop iterates on all the elements of the array where the counter for each dimension is found in the same way as explained in Section 8.6.8. The position in the array is trivial since it is always the same as the overall counter, as no broadcasting is possible here. The position in the result array, however, must be calculated. For each position, the number is added to the value already stored in the result array. This requires that all the elements of the result array are initialized to zero.

8.6.12 `mins` function

```
void mins(Pandora *result, double(*fun)(double *),
    int nl, Pandora **elem, double *tmp2, int *tmp1,
    int *tfarr, int rank, int *dim, ...)
```

This function is used to calculate the smallest value found from evaluating the λ-function given. This function has many similarities with the `siter` function described in Section 8.6.10. The main difference is how the result from the λ-function is treated. In this case the function takes nine mandatory arguments and a variable argument list. The first argument is the `Pandora` struct used to store the result. The second argument is the λ-function to be called. The third argument tells how many `Pandora` structs that are given to the variable argument list. The next five arguments are temporary memory segments used in the function. As for the `siter` function a rank reduction is performed here as well, and therefore the rank of the full struct must be given. An array of the actual dimensions must be given as the last argument since the size of the dimensions can be changed runtime. The function body is more or less the same as the `siter` function, but with a few important distinctions: First, the result from the λ-function is always a scalar. The other distinction is how the result from the λ-function is treated. In the `siter` function, the result is added to the value already stored in the correct position in the result array. Here, the value returned from the λ-function is compared with a value which is the smallest found from the previous calls to the λ-function. The smallest of the two values is kept. When looping over all the elements, the smallest one is found. The final value is stored in the `Pandora` struct.

8.6.13 `convert` function

```
double convert(Storage *storage, Pandora *pan)
```

It is possible to calculate the conversion factor between internal units and external units, based upon the units of the element and the scaling of each base unit. The `convert` method operates on the array of exponents, and the corresponding unit to find the scaling factor. Inside each `Pandora` struct there is an array of exponents stored. These exponents will be combined with the corresponding base unit to calculate the conversion factor. The contribution from each base unit is multiplied together to find the final scaling factor, which is returned.

8.6.14 `freeStorage` function

```
void freeStorage(Storage *storage)
```

This function is used to free all the memory allocated by the allocation methods. The `Storage` struct, holding all the memory segments is given as an argument to the function. First all the temporary memory segments are freed, but as explained in Section 8.6.3, the `Pandora` structs are stored in different kinds of arrays, and the number of structs in each array is also stored inside the struct. This information is used to loop through these arrays, and free all the `Pandora` structs stored inside the `Storage` struct. When all the structs have been freed the array itself is freed.

8.6.15 `matinv` function

```
void matinv(Pandora *pan, Pandora *invpan,
    Pandora *pivot)
```

This function is only an interface between the rest of the library and the external matrix inverse function. If the inverse of a scalar is required, the result is calculated within this function, otherwise the information needed is gathered from the `Pandora` structs given to the function, and sent to an external function. The function takes three arguments in total. The first argument is the matrix to be inverted, the second is a `Pandora` struct to store the result, and the final argument is a `Pandora` struct to store the pivots. Many inversion functions work directly on the array to be inverted, thereby replacing the original array with its inverse. In this case the original array needs to be kept unchanged, since it also is used later in the code. The values of the original array are therefore copied to the struct which holds the inverse. The rest of this function depends on which external matrix inversion routine is called. Both the LAPACK functions dgetrf/dgetri (Anderson et al., 1999), and an in-house inversion algorithm have been used. The advantage of using our own inverse function is that no external installation is required.

8.7 Testing and Validation

Extensive testing was done on different levels and stages during the development of the RGrad language. This section covers the most important procedures used in the testing. First of all the expressions which are created when a model is implemented, or when expressions are created internally, must fulfill certain properties. It is then possible to eliminate some errors at an early stage. Second, as mentioned in Section 7.1, the unit consistency check will make sure that all expressions are internally consistent. This test will detect errors in the λ-function which could either be an error made by the user while implementing the model, or an error in the differentiation of the λ-function. The unit consistency check checks scalar operations, and no broadcasting errors will be detected by this check, as it only affects the λ-function. Another procedure for testing the differentiation of the λ-functions is to use third party software tools (Maple, Ginsh) and compare the results. Because of the complexity of the λ-functions, and because each program orders the elements of the λ-function differently, a direct comparison of the differentiated λ-functions is impossible. Therefore, only the numerical gradients of the complete model have been compared. Both the internal differentiation and the third party tools gave the same results within machine precision.

Errors in the λ-functions are relatively easy to detect, but errors in the broadcasting on the other hand, are very subtle and thereby hard to detect. Fortunately, there is a special structure that the expressions must fulfill and which can be exploited in an automatic code test: Whenever an expression is created it can be ensured that all the objects in the argument list have the same number of dimension objects. This eliminates the most severe errors which may else cause the generated C-code to crash because it attempts to read or write outside the allocated memory. Due to the complexity of the higher-order gradients it is impossible to verify the exported code by inspection, even for trivial examples, and numerical test must be relied on.

Prior to the present work, some of the models had already been implemented in Matlab with hand-coded first and second order gradients. For these models, the numerical results were compared, and found to be equal within the machine precision. Another method for testing the gradients is to compare the calculated gradient with numeric differentiation. This requires of course that the base function is correct, which must be assured by comparing the result with external calculations. The problem with numeric differentiation is that both roundoff errors and truncation errors reduce the resolution of the results. A three-point formula was used for the numeric differentiation. An example of a result from numerical differentiation compared with the symbolic gradients is shown in Listing 8.22.

```
A_T: 1.000000
```

```
 A_VA_T: 1.000000
 A_NA_T: 1.000000;1.000000;1.000000
A_TTA_T: 1.000000
A_TVA_T: 1.000000
```

Listing 8.22: Numerical differentiation gives the same results as the automatic gradient calculations with six digits resolution.

Another numeric test that can be performed on thermodynamic functions is to check that the Euler homogeneity properties of the function is as expected. As Listing 8.23 shows, both U and A live up to the expected results. More information about Euler homogeneous functions can be found in Modell and Reid (1983) and Haug-Warberg (2006), but in brief the homogeneity of a gradient is reduced with one if the function is differentiated with respect to a extensive variable, and remains the same if the function is differentiated with respect to an intensive variable.

```
    U: +1.0...00000
  U_S: +0.0...00056
  U_V: +0.0...00076
  U_N: +0.0...00039;+0.0...00035;+0.0...00058
 U_SS: -1.0...00000
 U_SV: -0.9...99944
    A: +1.0...00000
  A_T: +0.9...99978
  A_V: -0.0...00006
  A_N: -0.0...00000;-0.0...00001;-0.0...00001
 A_TT: +1.0...00000
 A_TV: +0.0...00020
```

Listing 8.23: Test of Euler homogeneity shows the expected results, ... means 0 or 9 repeated ten times.

Finally, the code was used to recalculate some problems with known answers. For example, in Chapters 3 and 4 the Soave-Redlich-Kwong equation of state was used to develop a phase stability test, and to locate critical and tricritical points. These calculations required an extensive use of both Legendre transforms and higher-order gradients. All the results were as expected, and it was concluded that this would have been virtually impossible with an erroneous code.

Chapter 9

Concluding Remarks

In this work a grammar for defining explicit expressions of multidimensional arrays has been developed. The grammar remains closed with respect to gradient calculations, which means that gradients to infinite order can be calculated using symbolic expressions. The grammar has been used to construct a language, called RGrad, in which it is possible to write complex thermodynamic models. The RGrad language adds its functionality on top of the Ruby language. This makes the implementation effort significantly smaller than writing our own parser. In fact, without utilizing a high-level language such as Ruby to write the input language this project would not have been possible within the given time frame.

Even though the current focus has been on thermodynamic models, the RGrad language can be used to formulate all kinds of models written solely within the grammar definition. The grammar is such that it maintains the structure of multidimensional arrays, and of gradients of higher order (>2). If these features are not requested then RGrad is not the right tool.

RGrad has also a built in unit consistency check, which makes it virtually impossible to write inconsistent models. The units check aids the programmer in writing consistent models, and it also guarantees that the automatically generated gradients are consistent with the model definition. As an extension to RGrad it is possible to calculate Legendre transforms of any order, and to calculate the gradients of the transforms with respect to both transformed and untransformed variables.

A model implemented in RGrad can be translated into standard ANSI C-code, which is exported to a standard file structure. This C-code relies on the implementation of a small computation library which implements the necessary calculation procedures. A general C-API which is common to all exported models has been

written to make the model portable. The API defines how the numbers should stored and retrieved from the computation kernel, and how to start and stop the calculations. The general C-API can easily be used to write interfaces to other external programs, but it has so far been used to define Ruby and Matlab interfaces only. This means on the other hand that all models implemented in RGrad can be used both in Ruby and in Matlab with the same execution kernel.

For demonstration purposes a few models generated with the RGrad software tool has been used for thermodynamic stability calculations and for critical point calculations. These calculations were done to make probable that the generated code is correct, and to show the usefulness of the Legendre transform in thermodynamic calculations. In the course of this work a novel method for determining whether a phase is globally stable phase has been developed. This method explores the entire thermodynamic space in the search for the most stable phase assembly. The global search is achieved by Legendre transforming all variables except two extensive degrees of freedom. The two remaining variables are used to span the thermodynamic space at constant intensive properties. The unnamed thermodynamic potential $X(S,V,\mu)$ was chosen for this purpose, since this potential has the same number of extensive variables regardless of the number of chemical components in the system.

Several different approaches for exploring the thermodynamic space has been investigated. Ordinary differential equations solvers* worked as expected, but were slow compared to a much simpler home-made explicit step approach. Two differential algebraic equation solvers† were tried in Matlab, but they were not able to solve the problem with desired accuracy or speed either. The conclusion is that the explicit step approach is the best. This approach combines two different methods, both based on the integration along a constant μ manifold: One simple method, and a more advanced method using a third-order Taylor expansion for predicting the next step. The more advanced method is required because a folding of the manifolds can happen, which makes it impossible to continue in the same principal direction as in the previous step. The explicit step method was tested for two different systems with known properties and the algorithm was able to detect all the minima, which means the entire thermodynamic space was effectively explored.

The potential $X(S,V,\mu)$ was also used to define critical and tricritical point criteria. The advantage of these criteria is that they are independent of the number of chemical components in the system, which means the implemented algorithm can be used to locate such points independent of the number of chemical compo-

*Fourth order Runge-Kutta and Runge-Kutta-Fehlberg

†ode15s and ode23s

nents. The algorithm which was implemented based on the critical and tricritical criteria converged to all the reported solutions. Based on these calculations it was concluded that the calculations of the higher-order gradients must necessarily be correct, since it is practically impossible to locate a tricritical point with erroneous gradients. However, the gradients have of course been checked numerically as well, and found to be correct within the precision of numerical differentiation.

As an interesting spin-off, the Taylor expansion from a converged critical point of a two-component system was used to predict the phase boundary connected to the critical point. This proved to be quite successful and it was possible to predict, and converge, the phase boundary for a large temperature interval from the information available at the critical point only.

9.1 Recommendations and Further Work

The grammar presented in Chapter 6 of this thesis is designed for implementing thermodynamic models. Thermodynamic models have a quite general structure, which is similar to other vector functions found in fluid mechanics and transport phenomena. The grammar is therefore from the author's point of view complete, with one important exception: It is entirely possible to define more operations for the λ-functions. The only restriction for the operations inside a λ-function is that the operator must respond to differentiation, and that the operand must be scalar.

It might be tempting to add cycling of the dimensions (e.g. the transposition of matrices) to the grammar, but this is not recommended, because it opens for unpredictable runtime bugs which would be very hard to detect. If this feature is required in the modelling, it is always possible to alter the expressions such that the cycling is not needed.

An automatic Legendre transform feature was built on top of the RGrad language presented in Chapter 7. The Legendre transform has nothing to do with the RGrad language, but since it was needed for calculations it was decided to implement it on top of RGrad. Anyway, it was relatively easy to extend the program to allow this feature. Another natural extension would be to implement Massieu functions as well, as these also require a thermodynamic potential and its gradients. It might be better to implement these variable transformations directly in C, however, and let them work systematically on a full set of gradients. This allows for more flexible and faster runtime calculations of any of the transforms, provided that the gradients are available. Subsequent calls to the library would give transforms of higher order, and the same would be valid for gradient calculations of the transforms.

The advantage of implementing the transformations in C rather than in RGrad, is that transformations of single components can be made possible from C. In RGrad only the transformations of the complete component set is possible. Another advantage is that in RGrad all possible transformations need to be defined before the code is exported, while a C implementation of transformations can be done run-time. The disadvantage is that the interface to the transformations will be more complicated, since it is up to the user to give the correct gradients to the transformation function. In RGrad the operations can be done a priori, which shields the user from making mistakes. Any transformations made run-time are of course limited by the availability of an adequate set of gradients.

The thermodynamic models used in the previous chapters were first written in RGrad and then translated into C-code. It is possible to translate the models into other languages, as has been demonstrated in this thesis for Matlab and Ruby. The exportation requires of course that the calculation library is implemented in the new language as well. Unless there are special needs, however, it is recommended to use the C-API, which can easily be used to create interfaces to other languages and programs. Two natural extensions in this direction is to implement a COM interface as this covers many software tools on the Microsoft Windows platform, and to implement a CAPE-OPEN interface since this is created for the purpose of program interoperability, and is widely used by the process industry.

An interesting feature that was implemented at an early stage, but not maintained due to time constraints, is the ability to generate a closed set of binary expressions which calculates a specific result. This would enable very fast calculations of heat capacities, enthalpy, entropy, etc., as the overhead of the function calls is avoided. A method for collecting the desired results, and a function to wrap all the binary operations must then be implemented. This feature can be used to calculate any scalar derivative of the equation of state as a function of temperature, at given volume and composition. The compiler will also be able to optimize the calculations, and bring very fast calculations to the user.

The phase stability test presented in Chapter 3 can be used to decide whether a phase assembly is globally stable. To be effective, this method must be supported by a phase equilibrium calculation to help locate the equilibrium phases. The more stable phases found by the phase stability test can be used as starting points for calculating the stable phase assembly. The stability test in Chapter 3 uses two different explicit step approaches, where the solution method is changed dynamically to best solve the problem. The switch is based on heuristic rules, which unfortunately are not very robust. More work is therefore needed in order to provide a totally reliable phase stability test, but the results obtained so far are very promising.

Bibliography

Alberty, R. A. (1994). Legendre Transforms in Chemical Thermodynamics. *American Chemical Society*, 94(6):pp. 1457–1482.

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Croz, J. D., Greenbaum, A., Hammarling, S., McKenney, A. and Sorensen, D. (1999). *LAPACK Users' Guide*. SIAM, third ed.

Ascher, D., Dubois, P. F., Hinsen, K., Hugunin, J. and Oliphant, T. (2001). *An Open Source Project, Numerical Python*. Lawrence Livermore National Laboratory, Livermore, CA.

Baker, L. E., Pierce, A. C. and Luks, K. D. (1982). Gibbs Energy Analysis of Phase Equilibria. *Soc. Pet. Eng. J. (1980-)*, pp. 731–742.

Bartis, J. T. (1973). Thermodynamic Equations for Tri- or Third Order Critical Points. *J. Chem. Phys.*, 59(10):pp. 5423–5430.

Bauer, C., Frink, A. and Kreckel, R. (2002). Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language. *J. Symbolic Computation*, 33.

Beegle, B. L., Modell, M. and Reid, R. C. (1974a). Legendre Transformations and Their Applications in Thermodnyamics. *AIChE Journal*, 20(6):pp. 1194–1199.

Beegle, B. L., Modell, M. and Reid, R. C. (1974b). Thermodynamic Stability Criterion for Pure Substances and Mixtures. *AIChE J.*, 20(6):pp. 1200–1206.

Bischof, C., Carle, A., Corliss, G., Griewank, A. and Hovland, P. (1992a). ADIFOR – Generating Derivative Codes from Fortran Programs. *Scientific Programming*, 1(1):pp. 11–29.

Bischof, C. H., Corliss, G. F., Green, L., Griewank, A., Haigler, K. and Newman, P. (1992b). Automatic Differentiation of Advanced CFD Codes for Multidisciplinary Design. *Computing Systems in Engineering*, 3:pp. 625–637.

Branin, F. H., Jr. (1972). Widely Convergent Method for Finding Multiple Solutions of Simultaneous Nonlinear Equations. *IBM J. Res. Dev.*, pp. 504–522.

Brendsdal, E. (1999). Computation of Phase Equilibria in Fluid Mixtures. Ph.D. thesis, Norwegian University of Science and Technology.

Callen, H. B. (1985). *Thermodynamics and an Introduction to Thermostatistics*. John Wiley & Sons, Inc, second ed.

Church, A. (1932). A Set of Postulates for the Foundation of Logic. *Annals of mathematics*.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press.

CO-LaN (2006). *Thermodynamic and Physical Properties interface specification v1.1*.

Falkoff, A. and Iverson, K. (1973). The Design of APL. *IBM Journal of Research and Development*, 17(4):pp. 324–334.

Gibbs, J. W. (1876). On the equilibrium of heterogeneous substances. *Trans. Conn. Acad*, III(108). As reprinted in *The Scientific Papers of J. Willard Gibbs*, Vol. 1, Dover, New York (1961).

Gilbert, J. C., Vey, G. L. and Masse, J. (1991). La differentiation automatique de fonctions representes par des programmes. Rapports de Recherche N° 1557, INIRA, France.

Griewank, A. (1989). On Automatic Differentiation. In Iri, M. and Tanabe, K. (eds.), *Mathematical Programming: Recent Developments and Applications*, pp. 83–108. Kluwer Academic Publishers, Dordrecht.

Griewank, A. (2000). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA.

Griewank, A., Juedes, D. and Utke, J. (1996). ADOL–C, A Package for the Automatic Differentiation of Algorithms Written in C/C++. *ACM Trans. Math. Software*, 22(2):pp. 131–167.

Hanson, J. W., Caviness, J. S. and Joseph, C. (1962). Analytic Differentiation By Computer. *Communications of the ACM*, 5(6):pp. 349 – 355.

Hascoët, L. and Pascual, V. (2004). TAPENADE 2.1 User's Guide. Rapport technique 300, INRIA, Sophia Antipolis.

Haug-Warberg, T. (2006). *Den termodynamiske arbeidsboken*. Kolofon Forlag AS.

Haug-Warberg, T. (2008). Private communications.

Heidemann, R. A. and Khalil, A. M. (1980). The Calculation of Critical Points. *AIChE J.*, 26(5):pp. 769–779.

IBM (1994). *APL2 Programming: Language Reference*. International Business Machines Corporation.

ISO 8601:2004(E) (2004). *Data elements and interchange formats – Information interchange – Representation of dates and times*. International Organization for Standardization.

ISO/IEC 14977:1996(E) (1996). *Information Technology – Syntactic Metalanguage – Extended BNF*. International Organization for Standardization.

Iverson, K. E. (1991). A Personal View of APL. *IBM Systems Journal*, 30(4):pp. 582–593.

Juedes, D. W. (1991). A Taxonomy of Automatic Differentiation Tools. In Griewank, A. and Corliss, G. F. (eds.), *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pp. 315–329. SIAM, Philadelphia, PA.

Kalaba, R., Plum, T. and Tesfatsion, L. (1987). Automation of Nested Matrix and Derivative Operations. *Applied Mathematics and Computation*, 23:pp. 243–268.

Kedem, G. (1980). Automatic Differentiation of Computer-programs. *ACM Transactions on Mathematical Software*, 6(2):pp. 150–165.

Kilakos, A. and Kalitventzeff, B. (1993). A New Implementation for Analytical Derivatives of Thermodynamic Properties and Its Beneficial Application on Dynamic Simulation. *Comput. Chem. Eng.*, 17(5/6):pp. 441–450.

Kim, J. G., Hunke, E. C. and Lipscomb, W. H. (2006). Sensitivity Analysis and Parameter Tuning Scheme for Global Sea-Ice Modeling. *Ocean Modeling Journal*, 14(1–2):pp. 61–80.

Kleene, S. C. (1935). A Theory of Positive Integers in Formal Logic. *American journal of mathematics*, 57:pp. 153–173, 219–244.

Knobler, C. M. and Scott, R. L. (1984). Multicritical Point in Fluid Mixtures: Experimental Studies. In *Fluid Phase Equilibria*, vol. 9. Academic Press.

Kohse, B. F. and Heidemann, R. A. (1993). Computation of Tricritical Points in Ternary Systems. *AIChE J.*, 39(7):pp. 1242–1257.

Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT*, 16:pp. 146–160.

Luenberger, D. G. (1984). *Linear and Nonlinear Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, second ed.

MathWorks (2008). *MATLAB 7, C and Fortran API Reference*.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, Part I. *Comm. Assoc. Mach*, 3:pp. 184–195.

McDonald, C. M. and Floudas, C. A. (1995). Global Optimization for the Phase and Chemical Equilibrium Problem: Application to the NRTL Equation. *Comput. Chem. Eng.*, 19(11):pp. 1111–1139.

Michelsen, M. L. (1982a). The Isothermal Flash Problem. Part I Stability. *Fluid Phase Equilibria*, 9:pp. 1–19.

Michelsen, M. L. (1982b). The Isothermal Flash Problem. Part II. Phase-Split Calculation. *Fluid Phase Equilib.*, 9:pp. 21–40.

Michelsen, M. L. (1984). Calculation of Critical Points and Phase Boundaries in the Critical Region. *Fluid Phase Equilib.*, 16:pp. 57–76.

Michelsen, M. L. (1986). Some Aspects of Multiphase Calculations. *Fluid Phase Equilib.*, 30:pp. 15–29.

Mischler, C., Joulia, X., Hassold, E., Galligo, A. and Esposito, R. (1995). Automatic Differentiation applications to computer aided process engineering. *Computers & Chemical Engineering*, 19:pp. 779–784.

Modell, M. and Reid, R. C. (1983). *Thermodynamics and its Applications*. Prentice-Hall, Englewood Cliffs, N.J, second ed.

Nolan, J. F. (1953). Analytical Differentiation on a Digital Computer. Master's thesis, Mass. Inst. Technology.

P. Heimbach and C. Hill and R. Giering (2005). An efficient exact adjoint of the parallel MIT general circulation model, generated via automatic differentiation. *Future Generation Computer Systems*, 21(8):pp. 1356–1371.

Rall, L. B. (1981). *Automatic Differentiation: Techniques and Applications*, vol. 120 of *Lecture Notes in Computer Science*. Springer, Berlin.

Reid, R. C. and Beegle, B. L. (1977). Critical Point Criteria in Legendre Transform Notation. *AIChE Journal*, 23:pp. 726–731.

Soave, G. (1972). Equilibrium Constants from a Modified Redlich-Kwong Equation of State. *Fluid Phase Equilib.*, pp. 1197–1203.

Speelpenning, B. (1980). Compiling Fast Partial Derivatives of Functions Given by Algorithms. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL.

Sun, A. C. and Seider, W. D. (1995). Homotopy-Continuation Method for Stability Analysis in the Glob al Minimization of the Gibbs Free Energy. *Fluid Phase Equilib.*, 103:pp. 213–249.

Taylor, R. (1997). Automatic derivation of thermodynamic property functions using computer algebra. *Fluid Phase Equilibria*, 129:pp. 37–47.

Taylor, R. (1998a). Thermodynamics with Maple: I - Symbolic Computation. *Mathematics and Computers in Simulation*, 45:pp. 101–119.

Taylor, R. (1998b). Thermodynamics with Maple.: II - Numerical and Graphical Applications. *Mathematics and Computers in Simulation*, 45:pp. 121–146.

Thiéry, R. (1996). A new object-oriented library for calculating analytically high-order multivariable derivatives and thermodynamic properties of fluids with equations of state. *Computers & Geosciences*, 22(7):pp. 801–815.

Thom, R. (1975). *Structural Stability and Morphogenesis*. W.A. Benjamin, Inc, Reading MA.

Thomas, D., Fowler, C. and Hunt, A. (2005). *Programming Ruby: The Pragmatic Programmer's Guide*. The Pragmatic Programmer LLC, second ed.

van Konynenburg, P. H. and Scott, R. L. (1980). Critical Lines and Phase Equilibria in Binary van der Waals Mixtures. *Philos. Trans. R. Soc. London, Ser. A*, 298(1442):pp. 495–540.

Wengert, R. (1964). A Simple Automatic Derivative Evaluation Program. *Communications of the ACM*, 7(8):pp. 463–464.

Willhoft, R. G. (1991). Parallel expression in the APL2 language. *IBM Systems Journal*, 30(4):pp. 498–512.

Wilson, G. M. (1964). Vapor-Liquid Equilibrium. XI. A New Expression for the Excess Free Energy of Mixing. *J. Am. Chem. Soc.*, 86:pp. 127–130.

Zeeman, E. (1977). *Catastrophe Theory. Selected Papers 1972-1977*. Addison-Wesley Publishing Company, Inc., Reading MA.

Appendix A

Higher Order Gradients of the Legendre Transform

This appendix first summarise the derivations done in Section 2.1, and then shows the third order gradient of the Legendre transform.

The function definitions

A function is defined as $f(r,t)$, where r corresponds to a set of variables that will be Legendre transformed, and t corresponds to a set of variables that will not be transformed. The Legendre transform is defined as:

$$g = f - f_r r = f - \sigma r \tag{A.1}$$

$$\sigma \equiv f_r \tag{A.2}$$

It is now possible to express the function in two different coordinate systems, where $h(\sigma, \theta)$ corresponds to the transformed coordinate system, and $h(r,t)$ is the original. Gradients of the transformed function need to be expressed in the original coordinates. The following conversion rules can be used:

$$h_\sigma f_{rr}\,\mathrm{d}r = h_r\,\mathrm{d}r$$

$$h_\theta\,\mathrm{d}t + h_\sigma f_{tr}\,\mathrm{d}t = h_t\,\mathrm{d}t$$

$$h_r = h_\sigma f_{rr} \tag{A.3}$$

$$h_t = h_\theta + h_\sigma f_{tr} \tag{A.4}$$

First Derivatives

$$g_r = f_r - f_{rr} r - f_r \delta = -f_{rr} r \tag{A.5}$$
$$g_t = f_t - f_{tr} r \tag{A.6}$$

When the conversion rules are applied, this yields:

$$g_\sigma f_{rr} = -f_{rr} r \tag{A.7}$$
$$g_\theta + g_\sigma f_{tr} = f_t - f_{tr} r \tag{A.8}$$

Combine and order:

$$g_\sigma = -f_{rr}^{-1} f_{rr} r = -\delta r = -r \tag{A.9}$$
$$g_\theta = f_t - f_{tr} r - g_\sigma f_{tr} = f_t - f_{tr} r + r f_{tr} = f_t \tag{A.10}$$

Second Derivatives

Derivatives of g_σ:

$$g_{r\sigma} = \delta \tag{A.11}$$
$$\text{Rule A.3: } g_{\sigma\sigma} f_{rr} = -\delta \tag{A.12}$$
$$g_{\sigma\sigma} = -f_{rr}^{-1} \tag{A.13}$$
$$\tag{A.14}$$

$$g_{t\sigma} = 0 \tag{A.15}$$
$$\text{Rule A.4: } g_{\theta\sigma} + g_{\sigma\sigma} f_{tr} = 0 \tag{A.16}$$
$$g_{\theta\sigma} = f_{rr}^{-1} f_{tr} \tag{A.17}$$

Derivatives of g_θ:

$$g_{r\theta} = f_{rt} \tag{A.18}$$
$$\text{Rule A.3: } g_{\sigma\theta} f_{rr} = f_{rt} \tag{A.19}$$
$$g_{\sigma\theta} = f_{rt} f_{rr}^{-1} \tag{A.20}$$

$$g_{t\theta} = f_{tt} \tag{A.21}$$

$$\text{Rule A.4: } g_{\theta\theta} + g_{\theta\sigma} f_{tr} = f_{tt} \tag{A.22}$$

$$g_{\theta\theta} = f_{tt} - f_{rr}^{-1} f_{tr} f_{tr} \tag{A.23}$$

Third Derivatives

Derivatives of $g_{\sigma\sigma}$

$$\text{Rule A.3: } g_{\sigma\sigma\sigma} f_{rr} = g_{r\sigma\sigma} \tag{A.24}$$

$$g_{r\sigma\sigma} f_{rr} + g_{\sigma\sigma} f_{rrr} = 0 \tag{A.25}$$

$$g_{\sigma\sigma\sigma} f_{rr} f_{rr} + g_{\sigma\sigma} f_{rrr} = 0 \tag{A.26}$$

$$g_{\sigma\sigma\sigma} f_{rr} f_{rr} - f_{rr}^{-1} f_{rrr} = 0 \tag{A.27}$$

$$\text{Combine and order: } g_{\sigma\sigma\sigma} = f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} \tag{A.28}$$

$$\text{Rule A.4: } g_{\theta\sigma\sigma} + g_{\sigma\sigma\sigma} f_{tr} = g_{t\sigma\sigma} \tag{A.29}$$

$$g_{t\sigma\sigma} f_{rr} + g_{\sigma\sigma} f_{trr} = 0 \tag{A.30}$$

$$g_{\theta\sigma\sigma} f_{rr} + g_{\sigma\sigma\sigma} f_{tr} f_{rr} + g_{\sigma\sigma} f_{trr} = 0 \tag{A.31}$$

$$\text{Combine and order: } g_{\theta\sigma\sigma} = f_{rr}^{-1} f_{trr} f_{rr}^{-1} - f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} \tag{A.32}$$

Derivatives of $g_{\theta\sigma}$

$$\text{Rule A.3: } g_{\sigma\theta\sigma} f_{rr} = g_{r\theta\sigma} \tag{A.33}$$

$$g_{r\theta\sigma} + g_{r\sigma\sigma} f_{tr} + g_{\sigma\sigma} f_{rtr} = 0 \tag{A.34}$$

$$g_{\sigma\theta\sigma} f_{rr} + g_{\sigma\sigma\sigma} f_{rr} f_{tr} + g_{\sigma\sigma} f_{rtr} = 0 \tag{A.35}$$

$$\text{Combine: } g_{\sigma\theta\sigma} = f_{rr}^{-1} f_{rtr} f_{rr}^{-1} - f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} \tag{A.36}$$

$$\text{Rule A.4: } g_{\theta\theta\sigma} + g_{\sigma\theta\sigma} f_{tr} = g_{t\theta\sigma} \tag{A.37}$$

$$g_{t\theta\sigma} + g_{t\sigma\sigma} f_{tr} + g_{\sigma\sigma} f_{ttr} = 0 \tag{A.38}$$

$$g_{\theta\theta\sigma} + g_{\sigma\theta\sigma} f_{tr} + g_{\theta\sigma\sigma} f_{tr} + g_{\sigma\sigma\sigma} f_{tr} f_{tr} + g_{\sigma\sigma} f_{ttr} = 0 \tag{A.39}$$

$$\begin{aligned}\text{Combine: } g_{\theta\theta\sigma} = {} & f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} - f_{rr}^{-1} f_{rtr} f_{rr}^{-1} f_{tr} \\ & + f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} - f_{rr}^{-1} f_{trr} f_{rr}^{-1} f_{tr} \\ & - f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} + f_{rr}^{-1} f_{ttr}\end{aligned} \tag{A.40}$$

This simplifies to:

$$\begin{aligned}g_{\theta\theta\sigma} = {} & f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} - f_{rr}^{-1} f_{rtr} f_{rr}^{-1} f_{tr} \\ & - f_{rr}^{-1} f_{trr} f_{rr}^{-1} f_{tr} + f_{rr}^{-1} f_{ttr}\end{aligned} \tag{A.41}$$

Derivatives of $g_{\sigma\theta}$

$$\text{Rule A.3: } g_{\sigma\sigma\theta} f_{rr} = g_{r\sigma\theta} \tag{A.42}$$

$$g_{r\sigma\theta} f_{rr} + g_{\sigma\theta} f_{rrr} = f_{rrt} \tag{A.43}$$

$$g_{\sigma\sigma\theta} f_{rr} f_{rr} + g_{\sigma\theta} f_{rrr} = f_{rrt} \tag{A.44}$$

$$\text{Order and combine: } g_{\sigma\sigma\theta} = f_{rrt} f_{rr}^{-1} f_{rr}^{-1} - f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} \tag{A.45}$$

$$\text{Rule A.4: } g_{\theta\sigma\theta} + g_{\sigma\sigma\theta} f_{tr} = g_{t\sigma\theta} \tag{A.46}$$

$$g_{t\sigma\theta} f_{rr} + g_{\sigma\theta} f_{trr} = f_{trt} \tag{A.47}$$

$$g_{\theta\sigma\theta} f_{rr} + g_{\sigma\sigma\theta} f_{tr} f_{rr} + g_{\sigma\theta} f_{trr} = f_{trt} \tag{A.48}$$

$$\begin{aligned}\text{Order and combine: } g_{\theta\sigma\theta} = {} & f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} - f_{rrt} f_{rr}^{-1} f_{rr}^{-1} f_{tr} \\ & - f_{rt} f_{rr}^{-1} f_{trr} f_{rr}^{-1} + f_{trt} f_{rr}^{-1}\end{aligned} \tag{A.49}$$

Derivatives of $g_{\theta\theta}$

$$\text{Rule A.3: } g_{\sigma\theta\theta} f_{rr} = g_{r\theta\theta} \tag{A.50}$$

$$g_{r\theta\theta} + g_{r\sigma\theta} f_{tr} + g_{\sigma\theta} f_{trt} = f_{rtt} \tag{A.51}$$

$$g_{\sigma\theta\theta} f_{rr} + g_{\sigma\sigma\theta} f_{rr} f_{tr} + g_{\sigma\theta} f_{trt} = f_{rtt} \tag{A.52}$$

$$\begin{aligned}\text{Order and combine } g_{\sigma\theta\theta} = {} & f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} - f_{rrt} f_{rr}^{-1} f_{tr} f_{rr}^{-1} \\ & - f_{rt} f_{rr}^{-1} f_{trt} f_{rr}^{-1} + f_{rtt} f_{rr}^{-1}\end{aligned} \tag{A.53}$$

Rule A.4: $g_{\theta\theta\theta} + g_{\sigma\theta\theta} f_{tr} = g_{t\theta\theta}$ (A.54)

$$g_{t\theta\theta} + g_{t\sigma\theta} f_{tr} + g_{\sigma\theta} f_{ttr} = f_{ttt} \quad \text{(A.55)}$$

$$g_{\theta\theta\theta} + g_{\sigma\theta\theta} f_{tr} + g_{\theta\sigma\theta} f_{tr} + g_{\sigma\sigma\theta} f_{tr} f_{tr} + g_{\sigma\theta} f_{ttr} = f_{ttt} \quad \text{(A.56)}$$

$$\begin{aligned} g_{\theta\theta\theta} = {} & f_{ttt} - f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} \\ & + f_{rrt} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} + f_{rt} f_{rr}^{-1} f_{trt} f_{rr}^{-1} f_{tr} - f_{rtt} f_{rr}^{-1} f_{tr} \\ & - f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} + f_{rrt} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} \\ & + f_{rt} f_{rr}^{-1} f_{trr} f_{rr}^{-1} f_{tr} + f_{trt} f_{rr}^{-1} f_{tr} \\ & - f_{rrt} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} + f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{rr}^{-1} f_{tr} f_{tr} - f_{rt} f_{rr}^{-1} f_{ttr} \end{aligned} \quad \text{(A.57)}$$

This simplifies to :

$$\begin{aligned} g_{\theta\theta\theta} = {} & f_{ttt} - f_{rt} f_{rr}^{-1} f_{rrr} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} \\ & + f_{rrt} f_{rr}^{-1} f_{tr} f_{rr}^{-1} f_{tr} + f_{rt} f_{rr}^{-1} f_{trt} f_{rr}^{-1} f_{tr} - f_{rtt} f_{rr}^{-1} f_{tr} \\ & + f_{rt} f_{rr}^{-1} f_{trr} f_{rr}^{-1} f_{tr} - f_{trt} f_{rr}^{-1} f_{tr} - f_{rt} f_{rr}^{-1} f_{ttr} \end{aligned} \quad \text{(A.58)}$$

Appendix B

SRK Model Parameters

This appendix shows the parameters used inside the thermodynamic models applied in Chapters 3 and 4.

System CH_4–H_2S

Table B.1: Parameters

	CH_4	H_2S
T_c [K]	190.6	373.2
P_c [10^5 Pa]	45.4	89.46
m []	0.4973	0.6281
H_0 [kJ/mol]	-74.9	-20.18
S_0 [J/K mol]	186.27	205.6
C_1 [J/K mol]	19.25	31.94
C_2 [J/K^2 mol]	0.0521	0.0014
C_3 [10^{-5} J/K^3 mol]	1.1970	2.4320
C_4 [10^{-8} J/K^4 mol]	-1.1320	-1.1760e

Table B.2: Binary interaction parameters

	CH_4	H_2S
CH_4	0	0.1
H_2S	0.1	0

System CH_4–nC_6–H_2S

Table B.3: Parameters

	CH_4	nC_6	H_2S
T_c [K]	190.4	507.5	373.2
P_c [10^5 Pa]	46	30.1	89.4
m []	0.5020	09351	0.6343
H_0 [kJ/mol]	-74.9	-167.3	-20.18
S_0 [J/K mol]	186.27	388.74	205.6
C_1 [J/K mol]	19.25	-4.413	31.94
C_2 [J/K^2 mol]	0.0521	0.582	0.0014
C_3 [10^{-5} J/K^3 mol]	1.1970	-31.1900	2.4320
C_4 [10^{-8} J/K^4 mol]	-1.1320	6.4940	-1.1760

Table B.4: Binary interaction parameters

	CH_4	nC_6	H_2S
CH_4	0	0	0.08
nC_6	0	0	0.05
H_2S	0.08	0.05	0

Appendix C

Code Examples

This appendix will show the code of the Wilson model (Wilson, 1964), generated with the tool described in Part II. The code has been indented by an external program, and long lines have been automatically divided by LaTeX, otherwise the code is presented unaltered.

Model RGrad file

```
rgrad = RGrad::graph
rgrad.set_timestamp("2008-01-01T00:00:00Z")
rgrad.set_helpstring(%Q{\
This is an implementation of the Wilson model.
The model is differentiated with respect to t and n.
This is Legendre transformed with respect to n.\
})
nc    = RGrad::dim.init(3)

t      = RGrad::var.init(298.15).units!({"unit_temperature"=>1}).label('
    temperature')
n      = RGrad::var(nc).init([0.2, 0.3, 0.5]).units!({"unit_amount"=>1}).
    label('mole_numbers')

RGAS   = RGrad::fix.init(8.3145).label("fix_rgas").units!({"unit_time
    "=>-2, "unit_length"=>2, "unit_amount"=>-1, "unit_temperature"=>-1, "
    unit_mass"=>1})
ntot   = n.sum
lij    = RGrad::par(nc,nc).init([
          [1, 2, 3],
          [3, 1, 2],
          [2, 3, 1]]).label("par_lij")
rgrad  << RGrad::halt(n){|_n|_n}.min!.label('halt_negn')
lm     = RGrad::expr(lij,n[nil,nc]){|_lij,_mj|_lij*_mj}.sum!(nc,nil)
```

```
llm    = RGrad::expr(lij,n[nc,nil],lm[nc,nil]){|_lij,_mi,_lmi|_lij*_mi/
    _lmi}.sum!(nil,nc)
rgrad<< RGAS
mu     = RGrad::expr(t[nil],ntot[nil],lm,llm){|_t,_ntot,_lm,_llm|RGAS*_t
    *(1-(_lm/_ntot).log-_llm)}

euler = RGrad::expr(n,mu){|_n,_mu|_n*_mu}.sum!.grad!(n,mu)

flg = RGrad::flag.label('flag_2_derivatives')
rgrad<< RGrad::goto(flg).label('goto_2_derivatives')

rgrad<< euler.label('g')
rgrad<< RGrad::flag.label('flag_1_derivatives')
rgrad<< euler.grad(t).label('g_t')
rgrad<< euler.grad(n).label('g_n')

rgrad<< flg
rgrad<< euler.grad(t).grad(t).label('g_tt')
rgrad<< euler.grad(n).grad(n).label('g_nn')
rgrad<< euler.grad(n).grad(t).label('g_nt','g_tn')


leg = euler.legendre(n).label('z')
rgrad << leg
rgrad << leg.grad(n).label('z_n')
rgrad << leg.grad(n).grad(n).label('z_nn')

rgrad.finalize
rgrad.export('path') if RGrad::C::load?
rgrad.export('path') if RGrad::CRuby::load?
rgrad.export('path') if RGrad::CMex::load?
rgrad.export('path') if RGrad::Latex::load?
rgrad.export('path') if RGrad::Dot::load?
```

Model header file

```
#include "rgradapi.h"
#define WILSON_DEFAULT_RETURN_FLAG 0
#define WILSON_MOLE_NUMBERS 1
#define WILSON_HALT_NEGN 2
#define WILSON_FIX_RGAS 3
#define WILSON_GOTO_2_DERIVATIVES 4
#define WILSON_TEMPERATURE 5
#define WILSON_PAR_LIJ 6
#define WILSON_G_N 7
#define WILSON_G 8
#define WILSON_FLAG_1_DERIVATIVES 9
#define WILSON_G_NT 10
#define WILSON_G_TN 10
#define WILSON_G_T 11
#define WILSON_FLAG_2_DERIVATIVES 12
#define WILSON_G_TT 13
#define WILSON_G_NN 14
#define WILSON_Z 15
#define WILSON_Z_N 16
#define WILSON_G_NN_INV 17
```

```
#define WILSON_G_NN_PIVOTS 18
#define WILSON_Z_NN 19
#define WILSON_UNIT_AMOUNT 20
#define WILSON_UNIT_LENGTH 21
#define WILSON_UNIT_MASS 22
#define WILSON_UNIT_TEMPERATURE 23
#define WILSON_UNIT_TIME 24
void wilson_init(RGradapi *api, int sized0, int *arrd0);
char *wilson_timestamp();
char *wilson_help();
```

Model main C-file

```
#include "wilson.h"
#include "pandora.h"
static double lambda12(double *);
static double lambda3(double *);
static double lambda0(double *);
static double lambda2(double *);
static double lambda10(double *);
static double lambda4(double *);
static double lambda9(double *);
static double lambda6(double *);
static double lambda5(double *);
static double lambda13(double *);
static double lambda11(double *);
static double lambda8(double *);
static double lambda7(double *);
static double lambda1(double *);
int wilson_call(void *ptr);
void wilson_init(RGradapi * api, int sized0, int *arrd0)
{
    Storage *storage;
    double v0[3] = { 0.2, 0.3, 0.5 };
    double p0[1] = { 8.3145 };
    double h1[1] = { 1.0 };
    double v1[1] = { 298.15 };
    double p1[9] = { 1.0, 2.0, 3.0, 3.0, 1.0, 2.0, 2.0, 3.0, 1.0 };
    double h2[1] = { 1.0 };
    double h3[1] = { 1.0 };
    double p2[1] = { 0.0 };
    double c0[1] = { 1.0 };
    double c1[1] = { 1.0 };
    double c2[1] = { 1.0 };
    double c3[1] = { 1.0 };
    double c4[1] = { 1.0 };
    int D0D0[2] = { 3, 3 };
    int D0[1] = { 3 };
    int u1[5] = { 1, 0, 0, 0, 0 };
    int u2[5] = { -1, 2, 1, -1, -2 };
    int u3[5] = { 0, 0, 0, 0, 0 };
    int u4[5] = { 0, 0, 0, 1, 0 };
    int u5[5] = { -1, 2, 1, 0, -2 };
    int u6[5] = { 0, 2, 1, 0, -2 };
    int u7[5] = { 0, 2, 1, -1, -2 };
    int u8[5] = { 0, 2, 1, -2, -2 };
```

```
int u9[5] = { -2, 2, 1, 0, -2 };
int u10[5] = { -1, 0, 0, 0, 0 };
int u11[5] = { 2, -2, -1, 0, 2 };
Dim *d0;

storage = mymalloc(sizeof(Storage));
storage->ne = 22;
storage->nv = 2;
storage->np = 3;
storage->ni = 2;
storage->nh = 4;
storage->nd = 1;
storage->nc = 5;
storage->e = mymalloc(sizeof(Pandora *) * 22);
storage->v = mymalloc(sizeof(Pandora *) * 2);
storage->p = mymalloc(sizeof(Pandora *) * 3);
storage->i = mymalloc(sizeof(Pandora *) * 2);
storage->h = mymalloc(sizeof(Pandora *) * 4);
storage->c = mymalloc(sizeof(Pandora *) * 5);
storage->d = mymalloc(sizeof(Dim *) * 1);
storage->z = mymalloc(sizeof(Pandora *) * 25);
storage->elem = mymalloc(sizeof(Pandora *) * 5);
storage->tmp1 = mymalloc(sizeof(int) * 3);
storage->tmp2 = mymalloc(sizeof(double) * 5);
storage->tfarr = mymalloc(sizeof(int) * 5 * 3);
storage->dim = mymalloc(sizeof(int) * 3);
storage->sfact = mymalloc(sizeof(int) * 3);
d0 = makeDim(sized0, arrd0);
storage->c[0] = makePandora(5, u3, 0, 1, NULL, c0);
storage->c[1] = makePandora(5, u3, 0, 1, NULL, c1);
storage->c[2] = makePandora(5, u3, 0, 1, NULL, c2);
storage->c[3] = makePandora(5, u3, 0, 1, NULL, c3);
storage->c[4] = makePandora(5, u3, 0, 1, NULL, c4);
storage->d[0] = d0;
storage->e[0] = allocPandora(5, u1, 1, d0);
storage->e[1] = allocPandora(5, u1, 0);
storage->e[2] = allocPandora(5, u3, 1, d0);
storage->e[3] = allocPandora(5, u5, 1, d0);
storage->e[4] = allocPandora(5, u6, 0);
storage->e[5] = allocPandora(5, u2, 1, d0);
storage->e[6] = allocPandora(5, u2, 1, d0);
storage->e[7] = allocPandora(5, u7, 0);
storage->e[8] = allocPandora(5, u9, 1, d0);
storage->e[9] = allocPandora(5, u3, 2, d0, d0);
storage->e[10] = allocPandora(5, u9, 1, d0);
storage->e[11] = allocPandora(5, u5, 0);
storage->e[12] = allocPandora(5, u10, 2, d0, d0);
storage->e[13] = allocPandora(5, u10, 2, d0, d0);
storage->e[14] = allocPandora(5, u10, 2, d0, d0);
storage->e[15] = allocPandora(5, u9, 2, d0, d0);
storage->e[16] = allocPandora(5, u6, 0);
storage->e[17] = allocPandora(5, u6, 0);
storage->e[18] = allocPandora(5, u1, 1, d0);
storage->e[19] = allocPandora(5, u11, 2, d0, d0);
storage->e[20] = allocPandora(5, u11, 1, d0);
storage->e[21] = allocPandora(5, u11, 2, d0, d0);
storage->h[0] = allocPandora(5, u1, 0);
```

```
    storage->h[1] = makePandora(5, u3, 0, 1, NULL, h1);
    storage->h[2] = makePandora(5, u3, 0, 1, NULL, h2);
    storage->h[3] = makePandora(5, u3, 0, 1, NULL, h3);
    storage->i[0] = makeeyePandora(5, u3, 0);
    storage->i[1] = makeeyePandora(5, u3, 2, d0, d0);
    storage->p[0] = makePandora(5, u2, 0, 1, NULL, p0);
    storage->p[1] = makePandora(5, u3, 2, 9, D0D0, p1, d0, d0);
    storage->p[2] = makePandora(5, u8, 0, 1, NULL, p2);
    storage->v[0] = makePandora(5, u1, 1, 3, D0, v0, d0);
    storage->v[1] = makePandora(5, u4, 0, 1, NULL, v1);
    storage->z[1] = storage->v[0];
    storage->z[2] = storage->h[0];
    storage->z[3] = storage->p[0];
    storage->z[4] = storage->h[1];
    storage->z[5] = storage->v[1];
    storage->z[6] = storage->p[1];
    storage->z[7] = storage->e[3];
    storage->z[8] = storage->e[4];
    storage->z[9] = storage->h[2];
    storage->z[10] = storage->e[6];
    storage->z[11] = storage->e[7];
    storage->z[12] = storage->h[3];
    storage->z[13] = storage->p[2];
    storage->z[14] = storage->e[15];
    storage->z[15] = storage->e[17];
    storage->z[16] = storage->e[18];
    storage->z[17] = storage->e[19];
    storage->z[18] = storage->e[20];
    storage->z[19] = storage->e[21];
    storage->z[20] = storage->c[0];
    storage->z[21] = storage->c[1];
    storage->z[22] = storage->c[2];
    storage->z[23] = storage->c[3];
    storage->z[24] = storage->c[4];
    api->call = &wilson_call;
    api->get = &get;
    api->set = &set;
    api->add_to_values = &add_to_values;
    api->get_number_of_elements = &get_number_of_elements;
    api->get_rank = &get_rank;
    api->get_dim = &get_dim;
    api->free = &destruct;
    api->ptr = (void *) storage;
    api->last_constant = WILSON_UNIT_TIME;
    api->memsize = getmemallocated();
    return;

}

int wilson_call(void *ptr)
{
    Storage *storage;
    Pandora **elem;
    int *tmp1;
    int *tfarr;
    int *dim;
    double *tmp2;
```

```
  int *sfact;
  Pandora **v;
  Pandora **p;
  Pandora **e;
  Pandora **i;
  Pandora **h;
  Dim **d;
  int tf[2] = { 1, 0 };
  int ttf[3] = { 1, 1, 0 };
  int ff[2] = { 0, 0 };
  int ftt[3] = { 0, 1, 1 };
  int tt[2] = { 1, 1 };
  int t[1] = { 1 };
  int tft[3] = { 1, 0, 1 };
  int f[1] = { 0 };
  int ft[2] = { 0, 1 };

  storage = (Storage *) ptr;
  elem = storage->elem;
  tmp1 = storage->tmp1;
  tfarr = storage->tfarr;
  dim = storage->dim;
  tmp2 = storage->tmp2;
  sfact = storage->sfact;
  v = storage->v;
  p = storage->p;
  e = storage->e;
  i = storage->i;
  d = storage->d;
  h = storage->h;
  mins(h[0], lambda0, 1, elem, tmp2, tmp1, tfarr, 1, dim, d[0], v[0], t)
      ;
  if (h[0]->arr[0] <= 0)
      return -WILSON_HALT_NEGN;
  if (h[1]->arr[0] <= 0)       /*goto_2_derivatives */
      goto flag_2_derivatives;
  siter(e[0], lambda1, 2, elem, tmp2, tmp1, tfarr, 2, dim, sfact, tf,
        d[0], d[0], p[1], tt, v[0], ft);
  sum(e[1], v[0], f, tmp1, sfact);
  siter(e[2], lambda2, 3, elem, tmp2, tmp1, tfarr, 2, dim, sfact, ft,
        d[0], d[0], p[1], tt, v[0], tf, e[0], tf);
  niter(e[3], lambda3, 4, elem, tmp2, tmp1, tfarr, v[1], f, e[0], t, e
      [1], f, e[2], t);       /*g_n */
  siter(e[4], lambda1, 2, elem, tmp2, tmp1, tfarr, 1, dim, sfact, f, d
      [0], v[0], t, e[3], t); /*g */
flag_1_derivatives:
  if (h[2]->arr[0] <= 0)
      return WILSON_FLAG_1_DERIVATIVES;
  niter(e[5], lambda4, 3, elem, tmp2, tmp1, tfarr, e[0], t, e[1], f,
        e[2], t);
  niter(e[6], lambda1, 2, elem, tmp2, tmp1, tfarr, e[5], t, i[0], f); /*
      g_nt, g_tn */
  siter(e[7], lambda1, 2, elem, tmp2, tmp1, tfarr, 1, dim, sfact, f, d
      [0], v[0], t, e[6], t); /*g_t */
flag_2_derivatives:
  if (h[3]->arr[0] <= 0)
      return WILSON_FLAG_2_DERIVATIVES;
```

```
    niter(e[8], lambda5, 2, elem, tmp2, tmp1, tfarr, v[1], f, e[0], t);
    siter(e[9], lambda1, 2, elem, tmp2, tmp1, tfarr, 3, dim, sfact, tft,
          d[0], d[0], d[0], p[1], ttf, i[1], ftt);
    niter(e[10], lambda6, 3, elem, tmp2, tmp1, tfarr, v[1], f, e[1], f,
          e[0], t);
    titer(e[11], lambda7, 1, elem, tmp2, v[1]);
    niter(e[12], lambda8, 2, elem, tmp2, tmp1, tfarr, p[1], tt, e[0], tf);
    niter(e[13], lambda9, 3, elem, tmp2, tmp1, tfarr, p[1], tt, v[0], tf,
          e[0], tf);
    siter(e[14], lambda10, 4, elem, tmp2, tmp1, tfarr, 3, dim, sfact, ftt,
          d[0], d[0], d[0], e[12], ttf, i[1], tft, e[13], ttf, e[9], tft);
    niter(e[15], lambda11, 5, elem, tmp2, tmp1, tfarr, e[8], tf, e[9], tt,
         e[10], tf, e[11], ff, e[14], tt);    /*g_nn */
    siter(e[16], lambda1, 2, elem, tmp2, tmp1, tfarr, 1, dim, sfact, f,
          d[0], e[3], t, v[0], t);
    titer(e[17], lambda12, 2, elem, tmp2, e[4], e[16]); /*z */
    titer(e[18], lambda13, 1, elem, tmp2, v[0]);        /*z_n */
    matinv(e[15], e[19], e[20]);
    titer(e[21], lambda13, 1, elem, tmp2, e[19]);       /*z_nn */
    return WILSON_DEFAULT_RETURN_FLAG;
}

static double lambda12(double *a)
{
    return a[0] - a[1];
}
static double lambda3(double *a)
{
    return 8.3145 * a[0] * (-(log(a[1] / a[2])) + 1.0 - a[3]);
}
static double lambda0(double *a)
{
    return a[0];
}
static double lambda2(double *a)
{
    return a[0] * a[1] / a[2];
}
static double lambda10(double *a)
{
    return a[0] * a[1] + a[2] * a[3];
}
static double lambda4(double *a)
{
    return 8.3145 * (-(log(a[0] / a[1])) + 1.0 - a[2]);
}
static double lambda9(double *a)
{
    return -(a[0] * a[1] * pow(a[2], -2.0));
}
static double lambda6(double *a)
{
    return 8.3145 * a[0] * -(pow(a[1], -2.0) * a[2] * -(a[1]) / a[2]);
}
static double lambda5(double *a)
{
    return 8.3145 * a[0] * -(1.0 / a[1]);
```

```
}
static double lambda13(double *a)
{
    return -(a[0]);
}
static double lambda11(double *a)
{
    return a[0] * a[1] + a[2] + a[3] * a[4];
}
static double lambda8(double *a)
{
    return a[0] / a[1];
}
static double lambda7(double *a)
{
    return -(8.3145 * a[0]);
}
static double lambda1(double *a)
{
    return a[0] * a[1];
}

char *wilson_timestamp()
{
    char *str = "2008-01-01T00:00:00Z";
    return str;
}

char *wilson_help()
{
    char *str = "This is an implementation of the Wilson model.\n\
The model is differentiated with respect to t and n.\n\
This is Legendre transformed with respect to n.";
    return str;
}
```

Model Ruby interface C-file

```
#include "wilson.h"
#include "ruby.h"
#define id_upcase rb_intern("upcase")
void rgrad_free(RGradapi * api);
VALUE klass_initialize(VALUE self, VALUE args);
VALUE pandoraclassptr;
VALUE classptr;
VALUE klass_get(VALUE, VALUE);
VALUE klass_set(VALUE, VALUE, VALUE);
VALUE klass_add_to_values(VALUE, VALUE, VALUE);
VALUE klass_call(VALUE);
VALUE klass_get_by_ptr(VALUE, VALUE, VALUE, VALUE, VALUE);
VALUE klass_set_by_ptr(VALUE self, VALUE cst,
                       VALUE memadr, VALUE memsize, VALUE offset);
VALUE klass_add_by_ptr(VALUE self, VALUE cst,
                       VALUE memadr, VALUE memsize, VALUE offset);
```

```
VALUE klass_initialize(VALUE self, VALUE args)
{
    int i;
    RGradapi *api;
    struct RArray *rb_carr;
    struct RArray *rb_carr2;
    api = malloc(sizeof(RGradapi));
    rb_carr = RARRAY(args);
    if ((int) rb_carr->len == 0) {
        int d0[3] = { 1, 1, 1 };
        wilson_init(api, 3, d0);
    } else {
        rb_carr2 = RARRAY(args);
        rb_carr = RARRAY(rb_carr2->ptr[0]);
        if ((int) rb_carr->len < 3)
            rb_raise(rb_eRuntimeError,
                     "Array need to be at least as big as original");
        int d0_size = (int) rb_carr->len;
        int *d0 = malloc(sizeof(int) * (int) rb_carr->len);
        for (i = 0; i < (int) rb_carr->len; i++)
            d0[i] = NUM2INT(rb_carr->ptr[i]);
        wilson_init(api, d0_size, d0);
        free(d0);
    }
    rb_iv_set(self, "@storage",
              Data_Wrap_Struct(pandoraclassptr, 0, rgrad_free, api));
    rb_iv_set(self, "@memsize", INT2FIX(api->memsize));
    return self;
}

void Init_wilson()
{
    VALUE apiclassptr;


    pandoraclassptr = rb_define_class("Pandoraswrap", rb_cObject);
    classptr = rb_define_class("Wilson", rb_cObject);
    apiclassptr = rb_define_class_under(classptr, "API", rb_cObject);
    rb_define_method(classptr, "initialize", klass_initialize, -2);
    rb_define_attr(classptr, "memsize", 1, 0);
    rb_define_method(classptr, "call", klass_call, 0);
    rb_define_method(classptr, "[]=", klass_set, 2);
    rb_define_method(classptr, "add_to_values", klass_add_to_values, 2);
    rb_define_method(classptr, "get_by_ptr", klass_get_by_ptr, 4);
    rb_define_method(classptr, "set_by_ptr", klass_set_by_ptr, 4);
    rb_define_method(classptr, "add_by_ptr", klass_add_by_ptr, 4);
    rb_define_method(classptr, "[]", klass_get, 1);
    rb_define_const(classptr, "HELP", rb_str_new2(wilson_help()));
    rb_define_const(classptr, "TIMESTAMP",
                    rb_str_new2(wilson_timestamp()));
    rb_define_const(apiclassptr, "MOLE_NUMBERS",
                    INT2FIX(WILSON_MOLE_NUMBERS));
    rb_define_const(apiclassptr, "HALT_NEGN", INT2FIX(WILSON_HALT_NEGN));
    rb_define_const(apiclassptr, "FIX_RGAS", INT2FIX(WILSON_FIX_RGAS));
    rb_define_const(apiclassptr, "GOTO_2_DERIVATIVES",
                    INT2FIX(WILSON_GOTO_2_DERIVATIVES));
    rb_define_const(apiclassptr, "TEMPERATURE",
```

```
                    INT2FIX(WILSON_TEMPERATURE));
    rb_define_const(apiclassptr, "PAR_LIJ", INT2FIX(WILSON_PAR_LIJ));
    rb_define_const(apiclassptr, "G_N", INT2FIX(WILSON_G_N));
    rb_define_const(apiclassptr, "G", INT2FIX(WILSON_G));
    rb_define_const(apiclassptr, "FLAG_1_DERIVATIVES",
                    INT2FIX(WILSON_FLAG_1_DERIVATIVES));
    rb_define_const(apiclassptr, "G_NT", INT2FIX(WILSON_G_NT));
    rb_define_const(apiclassptr, "G_TN", INT2FIX(WILSON_G_TN));
    rb_define_const(apiclassptr, "G_T", INT2FIX(WILSON_G_T));
    rb_define_const(apiclassptr, "FLAG_2_DERIVATIVES",
                    INT2FIX(WILSON_FLAG_2_DERIVATIVES));
    rb_define_const(apiclassptr, "G_TT", INT2FIX(WILSON_G_TT));
    rb_define_const(apiclassptr, "G_NN", INT2FIX(WILSON_G_NN));
    rb_define_const(apiclassptr, "Z", INT2FIX(WILSON_Z));
    rb_define_const(apiclassptr, "Z_N", INT2FIX(WILSON_Z_N));
    rb_define_const(apiclassptr, "G_NN_INV", INT2FIX(WILSON_G_NN_INV));
    rb_define_const(apiclassptr, "G_NN_PIVOTS",
                    INT2FIX(WILSON_G_NN_PIVOTS));
    rb_define_const(apiclassptr, "Z_NN", INT2FIX(WILSON_Z_NN));
    rb_define_const(apiclassptr, "UNIT_AMOUNT",
                    INT2FIX(WILSON_UNIT_AMOUNT));
    rb_define_const(apiclassptr, "UNIT_LENGTH",
                    INT2FIX(WILSON_UNIT_LENGTH));
    rb_define_const(apiclassptr, "UNIT_MASS", INT2FIX(WILSON_UNIT_MASS));
    rb_define_const(apiclassptr, "UNIT_TEMPERATURE",
                    INT2FIX(WILSON_UNIT_TEMPERATURE));
    rb_define_const(apiclassptr, "UNIT_TIME", INT2FIX(WILSON_UNIT_TIME));
    rb_eval_string
        ("class Wilson; def to_a_with_str(); return API.constants.sort.
            collect{|c| [c.downcase, self[c]]}; end; def to_a(); return
            API.constants.sort.collect{|c| self[c]}; end; end");
}

void rgrad_free(RGradapi * api)
{

    api->free(api->ptr);
    free(api);
    return;
}

VALUE klass_get_by_ptr(VALUE self, VALUE cst,
                       VALUE memadr, VALUE memsize, VALUE offset)
{
    int value, nbytes, ptrinc;
    RGradapi *api;
    struct RString *rb_cstr;
    char evalcode[31];
    size_t double_size;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
```

```
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        value = NUM2INT(rb_eval_string(evalcode));
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_get\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    double_size = sizeof(double);
    ptrinc = FIX2INT(offset);
    nbytes = api->get_number_of_elements(api->ptr, value) * double_size;
    if (nbytes > FIX2INT(memsize) - ptrinc * double_size)
        rb_raise(rb_eRuntimeError,
                 "The size of given array is too small\n");
    api->get(api->ptr, value, (double *) FIX2INT(memadr) + ptrinc);
    return self;
}

VALUE klass_get(VALUE self, VALUE cst)
{
    int i, value, nel;
    VALUE retval;
    RGradapi *api;
    struct RString *rb_cstr;
    char evalcode[31];
    double *tmparr;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        retval = rb_eval_string(evalcode);
        value = NUM2INT(retval);
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_get\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    nel = api->get_number_of_elements(api->ptr, value);
    if (api->get_rank(api->ptr, value) != 0) {
        retval = rb_ary_new2((long) nel);
        tmparr = malloc(sizeof(double) * nel);
        api->get(api->ptr, value, tmparr);
        for (i = 0; i < nel; i++)
            rb_ary_store(retval, i, rb_float_new(tmparr[i]));
    } else {
```

```
        tmparr = malloc(sizeof(double) * nel);
        api->get(api->ptr, value, tmparr);
        retval = rb_float_new(tmparr[0]);
    }
    free(tmparr);
    return retval;
}

VALUE klass_set(VALUE self, VALUE cst, VALUE val)
{
    int i, value, nel;
    struct RArray *rb_carr;
    struct RString *rb_cstr;
    char evalcode[31];
    RGradapi *api;
    VALUE tmp;
    double *tmparr;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        tmp = rb_eval_string(evalcode);
        value = NUM2INT(tmp);
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_get\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    nel = api->get_number_of_elements(api->ptr, value);
    switch (TYPE(val)) {
    case T_ARRAY:
        rb_carr = RARRAY(val);
        if (nel != (int) rb_carr->len)
            rb_raise(rb_eRuntimeError,
                     "Wrong size of array, %d given, should be %d\n",
                     (int) rb_carr->len, nel);
        tmparr = malloc(sizeof(double) * nel);
        for (i = 0; i < nel; i++)
            tmparr[i] = NUM2DBL(rb_carr->ptr[i]);
        api->set(api->ptr, value, tmparr);
        free(tmparr);
        break;
    case T_FIXNUM:
    case T_FLOAT:
        if (nel != 1)
            rb_raise(rb_eRuntimeError,
                     "Wrong number of elements, %d given, should be %d\n",
                     1, nel);
```

```
        tmparr = malloc(sizeof(double) * nel);
        tmparr[0] = NUM2DBL(val);
        api->set(api->ptr, value, tmparr);
        free(tmparr);
        break;
    default:
        rb_raise(rb_eRuntimeError, "Wrong type given\n");
    }
    return self;
}

VALUE klass_add_to_values(VALUE self, VALUE cst, VALUE val)
{
    int i, value, nel;
    struct RArray *rb_carr;
    struct RString *rb_cstr;
    char evalcode[31];
    RGradapi *api;
    VALUE tmp;
    double *tmparr;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        tmp = rb_eval_string(evalcode);
        value = NUM2INT(tmp);
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_get\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    nel = api->get_number_of_elements(api->ptr, value);
    switch (TYPE(val)) {
    case T_ARRAY:
        rb_carr = RARRAY(val);
        if (nel != (int) rb_carr->len)
            rb_raise(rb_eRuntimeError,
                     "Wrong size of array, %d given, should be %d\n",
                     (int) rb_carr->len, nel);
        tmparr = malloc(sizeof(double) * nel);
        for (i = 0; i < nel; i++)
            tmparr[i] = NUM2DBL(rb_carr->ptr[i]);
        api->add_to_values(api->ptr, value, tmparr);
        free(tmparr);
        break;
    case T_FIXNUM:
    case T_FLOAT:
        if (nel != 1)
```

```
            rb_raise(rb_eRuntimeError,
                     "Wrong number of elements, %d given, should be %d\n",
                     1, nel);
        tmparr = malloc(sizeof(double) * nel);
        tmparr[0] = NUM2DBL(val);
        api->add_to_values(api->ptr, value, tmparr);
        free(tmparr);
        break;
    default:
        rb_raise(rb_eRuntimeError, "Wrong type given\n");
    }
    return self;
}

VALUE klass_set_by_ptr(VALUE self, VALUE cst,
                       VALUE memadr, VALUE memsize, VALUE offset)
{
    int value, nbytes, panbytes, ptrinc;
    struct RString *rb_cstr;
    char evalcode[31];
    double *fromptr;
    RGradapi *api;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        value = NUM2INT(rb_eval_string(evalcode));
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_set_by_ptr\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    ptrinc = FIX2INT(offset);
    fromptr = ((double *) FIX2INT(memadr)) + ptrinc;
    nbytes = FIX2INT(memsize) - ptrinc * sizeof(double);
    panbytes =
        api->get_number_of_elements(api->ptr, value) * sizeof(double);
    if (panbytes > nbytes)
        rb_raise(rb_eRuntimeError, "Array given too small\n");
    api->set(api->ptr, value, fromptr);
    return self;
}

VALUE klass_add_by_ptr(VALUE self, VALUE cst,
                       VALUE memadr, VALUE memsize, VALUE offset)
{
    int value, nbytes, panbytes, ptrinc;
    struct RString *rb_cstr;
```

```
    char evalcode[31];
    double *fromptr;
    RGradapi *api;

    switch (TYPE(cst)) {
    case T_FIXNUM:
        value = NUM2INT(cst);
        break;
    case T_STRING:
        rb_cstr = RSTRING(rb_funcall(cst, id_upcase, 0));
        if (rb_cstr->len > 31)
            rb_raise(rb_eRuntimeError,
                     "Label name too long, use constant\n");
        sprintf(evalcode, "Wilson::API::%s", rb_cstr->ptr);
        value = NUM2INT(rb_eval_string(evalcode));
        break;
    default:
        rb_raise(rb_eRuntimeError, "Unknown type in klass_add_by_ptr\n");
    }
    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    if (api->last_constant < value || value < 1)
        rb_raise(rb_eRuntimeError, "constant out of range \n");
    ptrinc = FIX2INT(offset);
    fromptr = ((double *) FIX2INT(memadr)) + ptrinc;
    nbytes = FIX2INT(memsize) - ptrinc * sizeof(double);
    panbytes =
        api->get_number_of_elements(api->ptr, value) * sizeof(double);
    if (panbytes > nbytes)
        rb_raise(rb_eRuntimeError, "Array given too small\n");
    api->add_to_values(api->ptr, value, fromptr);
    return self;
}

VALUE klass_call(VALUE self)
{
    RGradapi *api;

    Data_Get_Struct(rb_iv_get(self, "@storage"), RGradapi, api);
    return INT2FIX((int) api->call(api->ptr));
}
```

Model mex interface C-file

```
#include "mex.h"
#include "wilson.h"

void mexFunction(int nlhs, mxArray * plhs[], int nrhs,
                 const mxArray * prhs[])
{
    RGradapi *api;
    double *pr;
    int i;
    if (nrhs == 0) {
        int d0[3] = { 1, 1, 1 };
        if (nlhs != 1)
            mexErrMsgTxt
```

```
                ("You must store the returned pointer, otherwise memory
                    will be lost\n");
        api = malloc(sizeof(RGradapi));
        wilson_init(api, 3, d0);
        plhs[0] = mxCreateDoubleScalar((int) api);
    } else if (mxIsNumeric(prhs[0])) {
        int *d0 = malloc(sizeof(int) * mxGetNumberOfElements(prhs[0]));
        pr = mxGetPr(prhs[0]);
        if (mxGetNumberOfElements(prhs[0]) < 3)
            mexErrMsgTxt
                ("You must give at least the same number of dimensions as
                    the default (3)\n");
        for (i = 0; i < mxGetNumberOfElements(prhs[0]); i++)
            d0[i] = pr[i];
        if (nlhs != 1)
            mexErrMsgTxt
                ("You must store the returned pointer, otherwise memory
                    will be lost\n");
        api = malloc(sizeof(RGradapi));
        wilson_init(api, 3, d0);
        plhs[0] = mxCreateDoubleScalar((int) api);
        free(d0);
    } else if (!strcmp("struct", mxArrayToString(prhs[0]))) {
        mxArray *structmatrix;
        const char *structnames[25] =
            { "fix_rgas", "flag_1_derivatives", "flag_2_derivatives", "g",
"g_n", "g_nn", "g_nn_inv", "g_nn_pivots", "g_nt", "g_t", "g_tn", "g_tt",
"goto_2_derivatives", "halt_negn", "mole_numbers", "par_lij", "temperature
    ",
"unit_amount", "unit_length", "unit_mass", "unit_temperature", "unit_time"
    , "z",
"z_n", "z_nn" };

        structmatrix = mxCreateStructMatrix(1, 1, 25, structnames);
        mxSetFieldByNumber(structmatrix, 0, 0,
                           mxCreateDoubleScalar(WILSON_FIX_RGAS));
        mxSetFieldByNumber(structmatrix, 0, 1,
                           mxCreateDoubleScalar
                           (WILSON_FLAG_1_DERIVATIVES));
        mxSetFieldByNumber(structmatrix, 0, 2,
                           mxCreateDoubleScalar
                           (WILSON_FLAG_2_DERIVATIVES));
        mxSetFieldByNumber(structmatrix, 0, 3,
                           mxCreateDoubleScalar(WILSON_G));
        mxSetFieldByNumber(structmatrix, 0, 4,
                           mxCreateDoubleScalar(WILSON_G_N));
        mxSetFieldByNumber(structmatrix, 0, 5,
                           mxCreateDoubleScalar(WILSON_G_NN));
        mxSetFieldByNumber(structmatrix, 0, 6,
                           mxCreateDoubleScalar(WILSON_G_NN_INV));
        mxSetFieldByNumber(structmatrix, 0, 7,
                           mxCreateDoubleScalar(WILSON_G_NN_PIVOTS));
        mxSetFieldByNumber(structmatrix, 0, 8,
                           mxCreateDoubleScalar(WILSON_G_NT));
        mxSetFieldByNumber(structmatrix, 0, 9,
                           mxCreateDoubleScalar(WILSON_G_T));
        mxSetFieldByNumber(structmatrix, 0, 10,
```

```
                         mxCreateDoubleScalar(WILSON_G_TN));
     mxSetFieldByNumber(structmatrix, 0, 11,
                         mxCreateDoubleScalar(WILSON_G_TT));
     mxSetFieldByNumber(structmatrix, 0, 12,
                         mxCreateDoubleScalar
                         (WILSON_GOTO_2_DERIVATIVES));
     mxSetFieldByNumber(structmatrix, 0, 13,
                         mxCreateDoubleScalar(WILSON_HALT_NEGN));
     mxSetFieldByNumber(structmatrix, 0, 14,
                         mxCreateDoubleScalar(WILSON_MOLE_NUMBERS));
     mxSetFieldByNumber(structmatrix, 0, 15,
                         mxCreateDoubleScalar(WILSON_PAR_LIJ));
     mxSetFieldByNumber(structmatrix, 0, 16,
                         mxCreateDoubleScalar(WILSON_TEMPERATURE));
     mxSetFieldByNumber(structmatrix, 0, 17,
                         mxCreateDoubleScalar(WILSON_UNIT_AMOUNT));
     mxSetFieldByNumber(structmatrix, 0, 18,
                         mxCreateDoubleScalar(WILSON_UNIT_LENGTH));
     mxSetFieldByNumber(structmatrix, 0, 19,
                         mxCreateDoubleScalar(WILSON_UNIT_MASS));
     mxSetFieldByNumber(structmatrix, 0, 20,
                         mxCreateDoubleScalar(WILSON_UNIT_TEMPERATURE));
     mxSetFieldByNumber(structmatrix, 0, 21,
                         mxCreateDoubleScalar(WILSON_UNIT_TIME));
     mxSetFieldByNumber(structmatrix, 0, 22,
                         mxCreateDoubleScalar(WILSON_Z));
     mxSetFieldByNumber(structmatrix, 0, 23,
                         mxCreateDoubleScalar(WILSON_Z_N));
     mxSetFieldByNumber(structmatrix, 0, 24,
                         mxCreateDoubleScalar(WILSON_Z_NN));
     plhs[0] = structmatrix;
 } else if (!strcmp("cell", mxArrayToString(prhs[0]))) {
     mxArray *tmpcell, *mycell;

     mycell = mxCreateCellMatrix(25, 1);
     tmpcell = mxCreateCellMatrix(2, 1);
     mxSetCell(tmpcell, 0, mxCreateString("mole_numbers"));
     mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_MOLE_NUMBERS));
     mxSetCell(mycell, 0, tmpcell);
     tmpcell = mxCreateCellMatrix(2, 1);
     mxSetCell(tmpcell, 0, mxCreateString("halt_negn"));
     mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_HALT_NEGN));
     mxSetCell(mycell, 1, tmpcell);
     tmpcell = mxCreateCellMatrix(2, 1);
     mxSetCell(tmpcell, 0, mxCreateString("fix_rgas"));
     mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_FIX_RGAS));
     mxSetCell(mycell, 2, tmpcell);
     tmpcell = mxCreateCellMatrix(2, 1);
     mxSetCell(tmpcell, 0, mxCreateString("goto_2_derivatives"));
     mxSetCell(tmpcell, 1,
               mxCreateDoubleScalar(WILSON_GOTO_2_DERIVATIVES));
     mxSetCell(mycell, 3, tmpcell);
     tmpcell = mxCreateCellMatrix(2, 1);
     mxSetCell(tmpcell, 0, mxCreateString("temperature"));
     mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_TEMPERATURE));
     mxSetCell(mycell, 4, tmpcell);
     tmpcell = mxCreateCellMatrix(2, 1);
```

```
mxSetCell(tmpcell, 0, mxCreateString("par_lij"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_PAR_LIJ));
mxSetCell(mycell, 5, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_n"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_N));
mxSetCell(mycell, 6, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G));
mxSetCell(mycell, 7, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("flag_1_derivatives"));
mxSetCell(tmpcell, 1,
          mxCreateDoubleScalar(WILSON_FLAG_1_DERIVATIVES));
mxSetCell(mycell, 8, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_nt"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_NT));
mxSetCell(mycell, 9, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_tn"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_TN));
mxSetCell(mycell, 10, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_t"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_T));
mxSetCell(mycell, 11, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("flag_2_derivatives"));
mxSetCell(tmpcell, 1,
          mxCreateDoubleScalar(WILSON_FLAG_2_DERIVATIVES));
mxSetCell(mycell, 12, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_tt"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_TT));
mxSetCell(mycell, 13, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_nn"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_NN));
mxSetCell(mycell, 14, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("z"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_Z));
mxSetCell(mycell, 15, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("z_n"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_Z_N));
mxSetCell(mycell, 16, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_nn_inv"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_NN_INV));
mxSetCell(mycell, 17, tmpcell);
tmpcell = mxCreateCellMatrix(2, 1);
mxSetCell(tmpcell, 0, mxCreateString("g_nn_pivots"));
mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_G_NN_PIVOTS));
mxSetCell(mycell, 18, tmpcell);
```

```
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("z_nn"));
        mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_Z_NN));
        mxSetCell(mycell, 19, tmpcell);
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("unit_amount"));
        mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_UNIT_AMOUNT));
        mxSetCell(mycell, 20, tmpcell);
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("unit_length"));
        mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_UNIT_LENGTH));
        mxSetCell(mycell, 21, tmpcell);
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("unit_mass"));
        mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_UNIT_MASS));
        mxSetCell(mycell, 22, tmpcell);
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("unit_temperature"));
        mxSetCell(tmpcell, 1,
                  mxCreateDoubleScalar(WILSON_UNIT_TEMPERATURE));
        mxSetCell(mycell, 23, tmpcell);
        tmpcell = mxCreateCellMatrix(2, 1);
        mxSetCell(tmpcell, 0, mxCreateString("unit_time"));
        mxSetCell(tmpcell, 1, mxCreateDoubleScalar(WILSON_UNIT_TIME));
        mxSetCell(mycell, 24, tmpcell);
        plhs[0] = mycell;
    } else if (!strcmp("help", mxArrayToString(prhs[0]))) {
        plhs[0] = mxCreateString(wilson_help());
    } else if (!strcmp("timestamp", mxArrayToString(prhs[0]))) {
        plhs[0] = mxCreateString(wilson_timestamp());
    } else {
        mexErrMsgTxt("Illegal string\n");
    }
    return;
}
```