NTNU – Trondheim
Norwegian University of
Science and Technology

Symbolic Differentiation of Multivariable Functions to Arbitrary Order

Kristian Selvaag

Chemical Engineering and Biotechnology
Submission date: June 2013
Supervisor: Tore Haug-Warberg, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

Summary

Thermodynamic properties, like pressure, volume and temperature can be calculated as partial derivatives of energy functions. Obtaining analytic partial derivatives can be tedious work. Computer algebra systems can be used instead. Taking the gradient of a multi-variable scalar function yields a vector, taking the gradient again yields a matrix, if higher order derivatives are desired, they would have to be organized in higher-dimensional algebraic structures. Support for these kinds of objects is limited in existing computer algebra systems.

The concept of a multidimensional algebraic object (MDO) is introduced: An MDO may have any number of dimensions. A scalar having zero dimensions, a vector one dimension and a matrix two dimensions. A lightweight computer algebra system has been implemented in Ruby. It enables the construction of mathematical expressions using MDOs as variables. These expressions can be both evaluated as functions and differentiated to return new MDO expressions.

The MDO code has been used to produce a phase diagram for a natural gas system modelled by the Redlich Kwong equation of state. To solve the implicit phase equilibrium equations, Newton's method was used: The objective function and Jacobian were evaluated as analytic functions derived by the MDO software - no numerical differentiation took place. The software provides explicit analytic derivatives to arbitrary order. Evaluation of high order derivatives, however, is time-consuming.

Preface

This thesis is written as a part of the Master's Degree program in Chemical Engineering at the Norwegian University of Science and Technology.

I would like to thank my supervisor Tore Haug-Warberg for providing guidance in a multitude of disciplines, spanning thermodynamics, mathematics and advanced programming. As a chemical engineering student, my object-oriented programming knowledge was limited, I had never heard of functional programming, or programmed in Ruby before, and my only experience with computer algebra systems was as a user. A steep learning curve followed. This would not have been overcome had it not been for my supervisor's eagerness to discuss issues and ideas along the way. With that said, I am also grateful for the freedom that I was given, which encouraged creativity in the programming process.

I declare that this is an independent work according to the exam regulations of the Norwegian University of Science and Technology (NTNU).

Name	Date

Contents

Roman letter symbols[1]

A	Helmholtz energy	J
A	A generic matrix	
A_i	Heat capacity parameter of component i	J/(molK)
a	Parameter in the Redlich Kwong equation of state	$\mathrm{PaK^{1/2}m^6}$
a	A generic variable	
a_i	Component parameter in the Redlich Kwong equation of state	$\mathrm{PaK^{1/2}m^6/mol^2}$
a_{ij}	Component pairwise parameter in the Redlich Kwong equation of state	$\mathrm{PaK^{1/2}m^6/mol^2}$
a^*	Broadcasted version of a	
$\hat{a}$	Another broadcasted version of a	
B_i	Heat capacity parameter of component i	$\mathrm{J/(molK^2)}$
b	Parameter in the Redlich Kwong equation of state	$\mathrm{m^3}$
b	A generic variable	
b_i	Component specific parameter in the Redlich Kwong equation of state	$\mathrm{m^3/mol}$
C	Total number of chemical components in a system	-
C_i	Heat capacity parameter of component i	$\mathrm{J/(molK^3)}$
c	A generic variable	
c_p	Molar heat capacity at constant pressure	J/(mol K)
D_i	Heat capacity parameter	$\mathrm{J/(molK^4)}$
d_0, d_1	Elements of the `dep` array	
E_i	Heat capacity parameter of component i	$\mathrm{J/(molK^5)}$
f	A generic function	
F	A generic function, used as the residual function $F(x) = 0$ in Newton iteration example	
G	Gibbs energy	J
g	Generic function	
$\mathbf{g}$	First derivative of total Helmholtz energy	
H	Hessian of total Helmholtz energy	
h	Molar enthalpy	J/(mol)
J	Jacobian matrix	

[1] Units are given in the right column. Where "-" appears, the quantity is unitless. If nothing is given in this field, the units are irrelevant (for example for generic symbols or symbols representing non-scalars)

k_{ij}	Interaction parameter between species i and j	-
l	Generic rank 2 MDO	
m	MDO rank	-
m	Generic rank 1 MDO	
N	Total number of moles in a system	mol
N	Total number of ways to broadcast from rank n to rank m	-
n	MDO rank	-
$\mathbf{n}$	Mole number vector	mol
n_i	Number of moles of component number i	mol
p	Pressure	Pa
p	Generic parameters	
S	Entropy	J/(K)
S	A structure operator	
s	Molar entropy	J/(mol K)
T	Temperature	K
U	Internal energy	J
V	Volume	m^3
v	The value returned by an evaluation call	
y	A generic variable	
$\mathbf{y}$	The partial derivatives of Helmholtz energy, but with constant temperature: $\left[\frac{\partial A}{\partial V}, \frac{\partial A}{\partial n_1}, \frac{\partial A}{\partial n_2}, ..., \frac{\partial A}{\partial n_c}\right]$	[J/m^3, J/mol, J/mol, ..., J/mol]
x	A generic variable, used as free variable in Newton iteration example	-
$\mathbf{x}$	The free variables of Helmholtz energy, except temperature: $[V, n_1, n_2, ..., n_C]$	[m^3, mol, mol, ..., mol]
z	A generic function	

Greek letter symbols

α	Generic scaling factor	-
γ	Generic 2-d MDO	
δ	Kronecker delta	-
Δ	Mathematical symbol denoting "change in". F. ex. Δh_0^f	
λ	Lambda function, a function that operates on scalar elements	
μ	Chemical potential	J/(mol)

Subscripts

0	Standard state (298 K, 10^5 Pa)
c	Critical
f	Formation
i	Index
$i_1 i_2 ... i_r$	Used for indexing a rank r MDO
j	Index
$j_1 j_2 ... j_r$	Used for indexing a rank r MDO
k	Index
l	Index
m	Index
tot	Total

Superscripts

0	Standard state (298 K, 10^5 Pa)
ig	Ideal gas
l	Liquid phase
res	Residual
RK	Redlich Kwong equation of state
$\top$	Matrix transpose
v	Vapour phase

Nomenclature

AD	Automatic Differentiation
BC	Broadcast
CAS	Computer Algebra System
child	A node is a child to a node which has a directed relation pointing at it
DAG	Directed Acyclic Graph
graph	Representation of linked objects
ig	Abbreviation denoting ideal gas equation of state
leaf	A graph node with no children
MDO	Multidimensional algebraic object
MDO object	An instance of the `MDO` class. Its structure can be represented by a graph: One MDO object links to other MDO objects
parent	A graph node is the parent of any node it has a directed relation to
RK-EOS	Abbreviation denoting the Redlich Kwong equation of state
rank	The number of dimensions of an MDO
root	A graph node without any parents

Chapter 1

Introduction

Given a thermodynamic model, usually an equation of state, it has been the traditional approach to manually derive analytic expressions for all kinds of partial derivatives needed in the calculations. With the emergence of new models of increasing complexity, such as the PC-SAFT equation of state (Gross and Sadowski, 2000), the traditional approach is becoming tedious and error prone.

One approach to the new challenges is to estimate derivatives *numerically*, using for example finite differences. While this can give good results for low order derivatives, it is prone to numerical errors as well as accumulated machine rounding errors (Jain, 2003).

A second approach is to use *Automatic Differentiation* (AD). Automatic differentiation is a technique for differentiating whole computer programs. The program must have one or several inputs as well as one or several outputs. Since each elementary operation in a program is differentiable, the chain rule can be applied to compute the derivative of the outputs with respect to the inputs. Mischler, Joulia, Hassold, Galligo, and Esposito (1995) applied AD to computer process engineering problems and report AD to be of comparable performance to implementation of the analytically derived derivatives.

Finally, the third approach is to use *symbolic differentiation*. This usually constitutes programming the model in a *computer algebra system* (CAS) and use it to derive desired derivative functions. Silva and Castier (1993) propose that using a computer algebra system helps solve the following problems in thermodynamics:

1. The derivation of relationships between thermodynamic properties.
2. Deriving expressions for activity coefficients given a thermodynamic model.
3. Differentiating complicated thermodynamic models to obtain partial derivatives for optimization problems like phase equilibrium calculations.

Points two and three can be addressed by AD as well. Another area where computer algebra is beneficial is implicit differentiation, which can be useful for estimating the sensitivities of iteratively found solutions to model parameters.

This paper provides documentation of work that has gone into solving the issue at the heart of the three problems mentioned above: Obtaining partial derivatives of thermodynamic energy functions (A, G, ...), from which all other properties can be derived. The *goal* of the investigation is to create a computer code capable of storing and differentiating function-expressions as well as organizing them into matrix-like algebraic objects (here called MDOs, more on this later), for use in thermodynamic modelling and equation solving. This goal might sound very general, but the intention of applying the code in the thermodynamic modelling of phase equilibrium has provided a focus for the design process.

The solution opted for in this paper was to implement a light-weight computer algebra system called `MDO`. Ruby was chosen as the implementation language. The unique feature of the `MDO` system is its general handling of multi-dimensional algebraic objects, be it vectors, matrices or higher-dimensional structures. These algebraic objects can be used to formulate expressions. Expressions can be differentiated to yield new expressions of the same type. These expressions can be differentiated again and so forth: Arbitrary order gradients can therefore be found. `MDO` expressions can also be evaluated as functions, thus they can be used in thermodynamic calculations.

Supporting theory on computer algebra as well as thermodynamic theory is presented in Chapter 2. The software implementation is covered in Chapter 3. The code has been successfully used to create a phase diagram for a natural gas system. This example also serves to demonstrate the syntax of the gradient derivation software, and can be found in Chapter 4. Overall, the project goals were met; namely algebraic gradients to arbitrary order could be found. However, issues that could be improved upon include performance (with regard to code execution speed) and flexibility. These issues are discussed in Chapter 5.

Chapter 2

Theory

This chapter is divided into five main parts: Sections 2.1 introduces the function representation most commonly used in computer algebra. This representation provides a base for gradient calculation software. Section 2.2 goes on to introduce the concept of multidimensional algebraic objects. Next, Section 2.3 provides some context to the choice of the programming style for the MDO software, at the heart of which, lies the gradient operator itself, discussed in Section 2.4. The final part, Section 2.5, outlines the theory of multicomponent phase equilibrium, and how this can be solved using the Redlich and Kwong (1949) equation of state and Newton's method.

2.1 Computer Algebra

In computer algebra, and computer science in general, graph representation of data is a useful concept. In this particular application, the *directed graph* (a graph with arrows) plays a special role. A directed graph is comprised of different parts, with different names. Figure 2.1 shows a graph labelled with terms used for its parts.

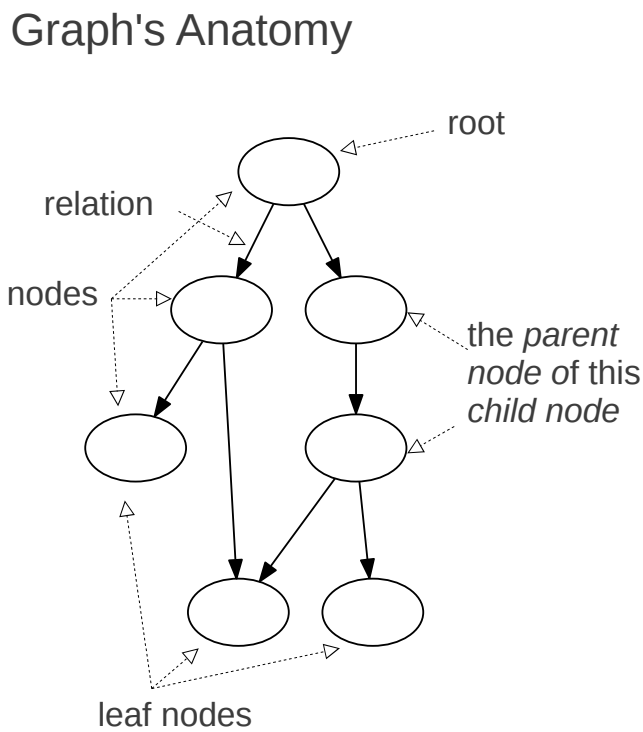

Figure 2.1: A graph with labels on each part. Every circle/ellipse is a *node*. The *root* node is the node without *parents*, while the arrows indicate which nodes are *parents* to which *child nodes*. Nodes without children are called *leaf* nodes, and the text inside each node is the node *label*.

The root node is at the *top*, while the children lie below, with the leaf nodes at the very *bottom*. Arrows between nodes are called *edges* or *relations*. The reader needs to have a clear grasp of the terminology related to graphs when reading the rest of this paper.

Function representation in computer algebra

One major difference between mathematical functions and functions encountered in programming is that mathematical functions themselves are often operated upon. Classical programming functions are usually defined, and then evaluated, whilst a mathematical function can be defined, differentiated, rearranged and combined with other mathematical functions, as well as evaluated. This calls for the use of a flexible data-structure representation of mathematical functions in their programming environment.

The most common way of representing a mathematical function in a computer algebra systems is *recursively* (Liska et al., 1999): Best visualized by a *directed acyclic graph* (DAG). The function

$$f(a, b) = ab + \ln(b/2) \tag{2.1}$$

can be represented by the graph shown in Figure 2.2. The arrows show the order

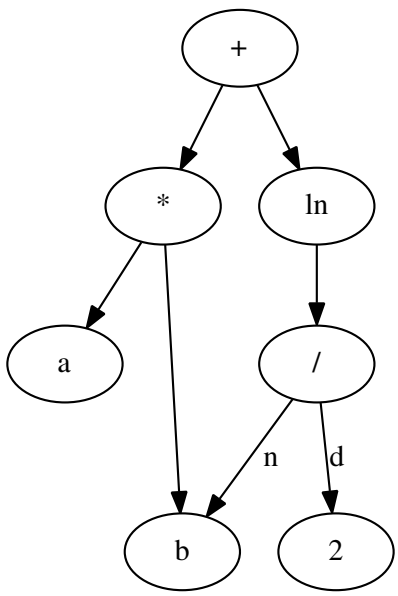

Figure 2.2: A graph representation of the function $f(a, b) = ab + \ln(b/2)$. Each node is either an operator or an operand (leaf node). The leaf nodes represent constants (f. ex. the node labelled 2) or named independent variables (a and b). The "n" and "d" on the arrows going out of the division node (labelled "/") denote numerator and denominator as the left and right operands respectively.

of operations: e.g. The top node (+) adds the values produced by its children, (*) and (ln), not the other way around. *Acyclic* means that it is impossible to return to a node by following a path of arrows leading from of it. The graph representation of the function in Equation (2.1) is not a *tree*, because nodes (like b) are allowed

to have multiple parents. In mathematical terms: The same variable can be used several times in an expression.

Evaluation of a function graph starts at the top node. The top node (+) requires the evaluation of *its* children, who in turn require evaluation of *their* children again, and so it goes on. When a leaf node is reached, it is either a constant (the node labelled 2 is an example of that), or an independent variable (nodes labelled a and b). When the evaluation algorithm reaches a leaf node, the value returned is simply the constant value, or that of the independent variable which was specified in the function call. The return values propagate back up the graph and are operated upon at each node on the way, until the function value itself is returned by the root node. How graphs are evaluated is covered in more detail later, in Section 3.3. Operations done to the function itself means a rearrangement of the graph, see Figure 2.3.

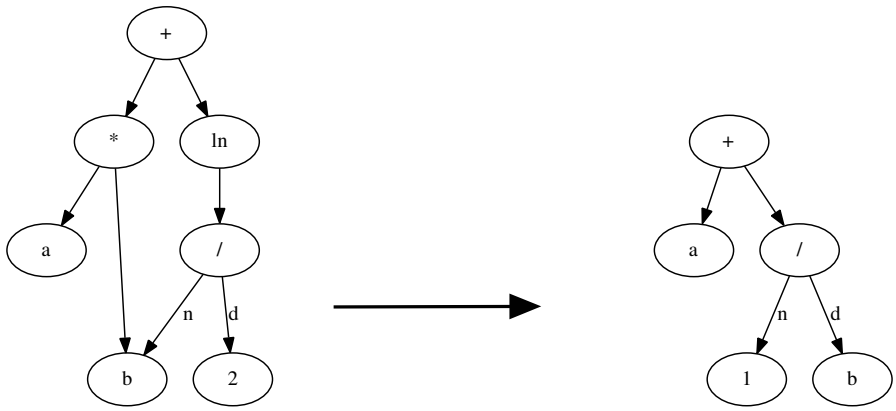

Figure 2.3: The function $f(a, b) = ab + \ln(b/2)$ is represented as a graph of operations on the left. The resulting graph to the right shows when f is differentiated with respect to b: $\partial f/\partial b = a + 1/b$.

Computer algebra and expression parsing A very similar concept to the expression graphs used in computer algebra is the *abstract syntax tree* used by compilers in syntax analysis (Lee, 2008). The parsing of expressions is not covered in this paper because the `MDO` computer algebra system uses operator overloading instead (more on this in Section 3.4). In practice, the Ruby interpreter itself does the parsing, and a new "language" did not have to be invented. Because no new language has been created, an Extended Backus-Naur Form meta-syntax has not been deemed beneficial for the reader's understanding of the `MDO` system and is therefore not included in this thesis.

2.2 Multidimensional (algebraic) Objects

Problems in equilibrium thermodynamics can often be represented as optimization problems, where the objective function $f(x)$ is an energy-potential to be minimized or maximized under certain constraints. Optimization problems can be formulated as $df/dx = 0$. To solve problems on this form, $F(x) = 0$, a Newton iteration scheme can be employed. Here $F(x) = df/dx$. The multivariate Newton iteration scheme is (Kelley, 2003):

$$x_{k+1} = x_k - J^{-1}(x_k)F(x_k) \tag{2.2}$$

J is the first derivative matrix, or *Jacobian*, of the function F. Since F is the first derivative of f, the Jacobian becomes the second derivative matrix, or *Hessian* of f. A thermodynamic system can, for example, have $1 + C$ degrees of freedom, where C is the number of components. Then the Hessian contains $(1 + C)^2$ elements, each being a scalar double derivative. The occurrence of matrices and vectors in thermodynamics calls for programs which are able to handle such concepts in a general and easy-to-use manner.

The area in which most computer algebra systems today fall short is in the treatment of vectors, matrices and higher order structures of functions, where the dimensions of these structures are not known beforehand, i.e. we want an implementation of a thermodynamic model where the component list can be specified by the user, and is not hard coded into the model. Taylor (1997) recognizes the need for this and gives the example of differentiating a sum where the upper limit is not yet specified. The implementation presented in this paper should include support for any function with unspecified dimension length.

To illustrate the need for this, take for example Helmholtz energy.

$$A(T, V, \mathbf{n}) \tag{2.3}$$

It is a function of $2 + C$ variables. Taking the gradient of this function produces a vector of $2 + C$ elements, taking the gradient again produces a matrix of $(2 + C) \times (2+C)$ elements and for each successive time the gradient is taken, a new dimension of size $(2+C)$ is added to the structure of the function object. In this work such an object will be called a Multi-Dimensional (algebraic) Object (MDO) and is taken to mean a collection of elements organized in a rectangular lattice of arbitrary rank. Rank here refers to the number of dimensions of an MDO. A vector would be represented by a rank 1 MDO, a matrix by a rank 2 MDO and a scalar by a rank 0 MDO. Calling the MDO a generalization of vectors and matrices is not accurate, as vectors and matrices have a sense of alignment thereby differentiating between rows and columns. To further illustrate the concept of MDOs, some examples are needed.

Figure 2.4 shows four different MDOs. The numbers contained in them have been chosen arbitrarily. The corresponding rank and size of its dimensions are listed below. To the far left in the figure, a rank 0 MDO is shown. It is similar to a scalar, while the rank 1 MDO to its right is similar to a vector. The rank 2 MDO is similar to a matrix, while the far right MDO is of rank 3 and does not have an

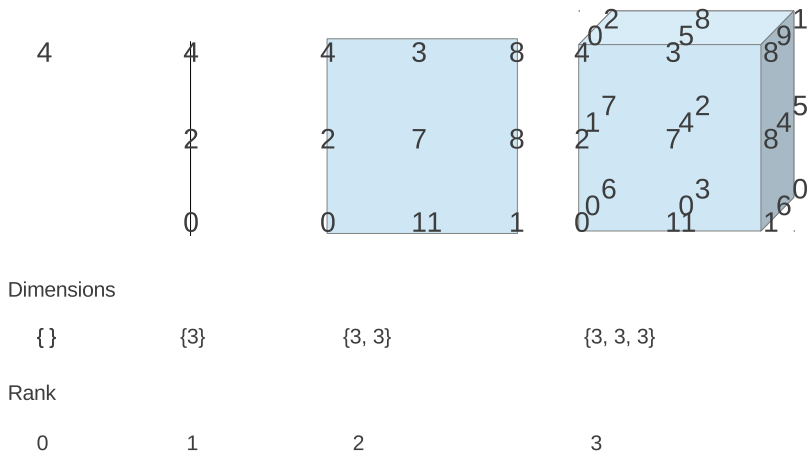

Figure 2.4: Four different MDOs represented as numbers distributed in several dimensions. Their dimension sizes are shown in curly brackets underneath, and at the bottom, their respective ranks are listed.

obvious analogy. Incidentally all dimension sizes were chosen to be three in this example.

The reason for getting rid of the idea of rows and columns is that it allows for more flexibility on the user's part. A vector-matrix product, for example, is not commutative, which means $Ax \neq xA$. Which elements are multiplied and what is summed is determined by whether the matrix or vector comes first in the multiplication. Multiplying two MDOs of different rank, the user would have to specify which elements align with each other. This does not apply to just MDO multiplication, but can be specified for an arbitrary function $\lambda(a, b, c...)$ of several MDOs and therefore provides great flexibility as to which elements combine in an expression.

The MDO, just like other algebraic objects, can be used as operands in mathematical expressions. A mathematical expression using MDOs as variables is called an *MDO expression*. Similarly, an *MDO function* is just like a mathematical function, but uses MDOs as arguments and outputs. Operations on MDOs are done element-by-element, but then the question arises: What if two MDOs of differing rank appear in the same MDO expression. Take, for example the MDO expression:

$$a + b \tag{2.4}$$

What if c was a rank 1 MDO, while b was a rank 2 MDO? There isn't an element in c for every element in b. The solution to this question is covered next.

Introduction to broadcasting

The term "broadcasting" is the name given to a technique for aligning an MDOs elements in several dimensions. It is best described through simple examples. Say that two MDOs a and b are to be added element-by-element. This could be denoted as

$$a + b \tag{2.5}$$

If a and b both have rank 0, they can be considered scalars and the above operation becomes trivial. If a and b both have the same rank, then element-by-element addition does just that: adds together the corresponding elements to produce a new MDO where each element equals the corresponding elements of a and b added together. Here is an example using $\text{rank}(a) = \text{rank}(b) = 2$

$$c_{ij} = a_{ij} + b_{ij} \tag{2.6}$$

The problems, however, arise when a and b do not have the same rank. A previous solution to this problem by Løvfall (2008) is to

1. Require that objects used in the same expression have the same dimensions as the highest rank object involved.

2. The rest of the objects copy their data into the missing dimensions to emulate a higher rank. The user specifies to which dimensions the data is copied, and in which dimensions to keep the original distribution.

Choosing which dimension to put the original data from and which dimensions to copy to is where broadcasting has to be specified. In this paper, broadcasting will work the same way, except that if MDOs of differing rank appear in the same expression, they will be broadcasted automatically with a default specification (discussed at the end of the next paragraph). This provides the benefit of a cleaner syntax for (much used) operations between scalars and MDOs of higher rank.

Take the above example, this time with $\text{rank}(a) = 1$ and $\text{rank}(b) = 2$. Following step number 1, the lambda function should assume all arguments to be of rank 2. The second step is to specify which elements of a to be used in combination which which elements in b. Call a^* the the broadcasted version of a, where $\text{rank}(a^*) = 2$. Expression (2.6) can now be evaluated as

$$c_{ij} = a^*_{ij} + b_{ij} \tag{2.7}$$

But a^* has to be specified. For this particular case there are two options:

$$a^*_{ij} = a_i \tag{2.8}$$

and

$$a^*_{ij} = a_j \tag{2.9}$$

In Equation (2.8), the elements of a have been distributed in the the first dimension (indexed by i) and copied into the second dimension (indexed by j), whereas in Equation (2.9) the data was laid out in the second dimension and copied into the first dimension. Coming back to the issue of *default* broadcasting: If an operator appears between MDOs of differing rank (no broadcasting is specified by the user), the lower rank MDO will be broadcasted with the added dimensions appearing *after* the existing ones, e.g. in the above example it would choose option (2.8). The difference between the two broadcasting specifications in Equations (2.8) and (2.9) has been visualized in Figure 2.5

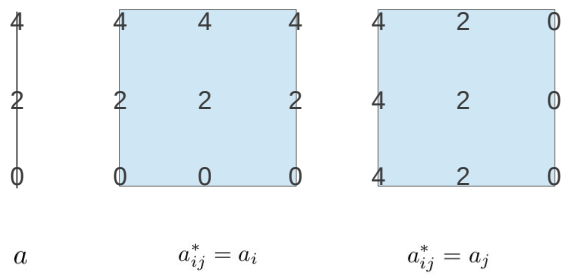

Figure 2.5: Example of different options when broadcasting a one-dimensional MDO $[4, 2, 0]$ to two dimensions. In this example i is the index of the first dimension (shown as vertical dimension), while j is the index of the second dimension (horizontal).

Higher rank broadcasting

Take a more complicated case: Say a needs to be broadcasted from rank 2 to rank 3. We have the following options:

$$a^*_{ijk} = a_{ij} \tag{2.10}$$

$$a^*_{ijk} = a_{ik} \tag{2.11}$$

$$a^*_{ijk} = a_{jk} \tag{2.12}$$

Other ways of taking a rank 2 MDO to rank 3 include:

$$a^*_{ijk} = a_{ji} \tag{2.13}$$

$$a^*_{ijk} = a_{ki} \tag{2.14}$$

$$a^*_{ijk} = a_{kj} \tag{2.15}$$

This covers all possible *ordered* combinations of two different elements from the set i, j, k. However, instead of including Equations (2.13) to (2.15) as broadcasting options, a new operation can be defined: Permutation, meaning the rearrangements of indices.

Permutation: Permutation[1] can be viewed as a generalization of the matrix transpose to include arbitrary rank MDOs. In a matrix transpose, elements of the matrix, a_{ij}, are rearranged according to

$$a_{ij}^{\top} = a_{ji} \tag{2.16}$$

The transposition operation shown in Equation (2.16) can be thought of the elements changing their location within the matrix. Another interpretation that the indices i and j switched places: $i, j \rightarrow j, i$. For a rank n MDO, there are $n!$ different ways of arranging the indices, which means that if you mean to permute them, there are $n! - 1$ options to choose from[2]. In order to achieve option (2.13) one has to do a permutation of $i, j \rightarrow j, i$ followed by broadcasting option (2.10). The need for Equations (2.13) to (2.15) as unique broadcasting options is eliminated, because they can be viewed as a combination of a permutation followed by a broadcast.

The total number N of possible combinations when broadcasting from rank n to rank m is given by the binomial coefficient because specifying a broadcasting scheme involves choosing n dimensions from a set of m dimensions, shown in Equation (2.17).

$$N = \binom{n}{m} \tag{2.17}$$

The binomial coefficients are visualized in Pascal's triangle in Figure 2.6: The position from the left indicates the original rank of the MDO, starting at rank 0. The row number in pascal's triangle corresponds to the rank to broadcast *to* (starting at 0 on the row containing just 1) . So if a rank 2 MDO was to be broadcasted to a rank 4, there are 6 specifications to choose from (5th row from the top, 3rd position from the left).

2.3 Programming

The Ruby language was chosen to implement the MDO gradient calculation system. The following sub-section shows that, among many programming styles, object-oriented and declarative programming would be the most helpful in creating the MDO computer algebra system. Ruby supports object-oriented programming and many other programming styles, as well as being dynamic and having a simple syntax (Flanagan and Matsumoto, 2008).

[1] GNU Octave is an example of a numerical computation software that implements both broadcasting and permutation (Eaton, 2011).

[2] Minus one, because there is always the default configuration. So if you mean to do a permutation there are $n! - 1$ *other* index configurations to choose from

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

Figure 2.6: Pascal's triangle:

Paradigms

Most programming languages support a variety of programming styles. Ruby is no exception. These programming styles can be loosely divided into programming 'paradigms'. Kedar (2008) lists five different programming paradigms:

1. Procedural (Imperative)
2. Functional (Applicative)
3. Logical (Rule-based)
4. Object-oriented
5. Concurrent

In a procedural program, code is executed as it is encountered, for example

```
int a = 1
int b = 2
int c = 3

d = a + c*b
```

Now, the value of `d` is 7. The state of the program (the existence and values of variables) keeps changing for each line. Typical examples of procedural languages are C and Fortran.

Functional programming, on the other hand, is more about passing data through functions, not keeping states in between. A similar program to the above written in a functional style might look like

```
c = add(1, multiply(2, 3))
```

Functional languages include, among others, Lisp and ML.

Not to be confused with the `if` and `switch` statements used in most languages, logical, or *rule based* programming, sets up a set of relations to control the steps in a computation. Much like in functional programming, the actual computations are done first when you *query* these relations. The prime example of a logical language is Prolog.

Object oriented programming aims at organizing data and code in a reusable manner. Data types, called *classes* encapsulate other data types. Objects of a class can be created, storing unique values of the encapsulated data. Consider a class with two member variables.

```
class Myclass
      a
      b
end
```

Objects can be created, and the class members, `a` and `b`, can be assigned values.

```
x = Myclass::new[3]
x.a = 1
x.b = 2

y = Myclass::new
y.a = 3
y.b = 4
```

The objects `x` and `y` keep their own copies of `a` and `b`: `a` could be `1` in one object, and `3` in another. C++ and Java are two major object-oriented languages.

Concurrent programming refers to programs where processes can happen simultaneously, for example in parallel computing. This is not applicable in the programming of the MDO[4]. *Object oriented programming*, however, plays a major role, as the MDO can be conveniently defined as its own class: `MDO`. The inner workings of the `MDO` class is described in Chapter 3

Another word that is often used about functional and logical programming is *declarative* programming (Chakravarty, 1997). The term is used when the steps in a computation are expressed without specifying the flow of the computation. For example in functional programming, functions are defined as separate steps. The steps are not executed until the functions are called. They may be called in any order, regardless of the order they were defined in.

There is a resemblance between declarative programming and mathematics: Mathematical expressions are defined first, and evaluated later. For this reason the usage of `MDO` objects will follow the style of declarative programming. The following pseudo-code is meant to illustrate the vision of how a light-weight computer algebra system could be implemented using ideas from declarative programming:

[3] `Class::new` is the default constructor in Ruby and returns an object of class `Class`

[4] Computing each element of an MDO in parallel could be highly beneficial for function evaluation performance, but it is beyond the scope of this paper

```
# Declaration of free variables
x1 = MDO::new
x2 = MDO::new
.
.
.

# Declaration of functions, no calculation goes on here
f1 = x1+x2
f2 = f1*x2+...
.
d1 = f1.grad(x2) # find gradients
d2 = d1.grad(x1)
.

# Initialization of parameters and free variable values
x1_val = 4.0
x2_val = [1, 2, 3]
.
.
.

# Evaluation:  The execution of the functions takes place here
f1.eval(x1_val, x2_val)
f2.eval(...)
d1.eval(...)
.
.
```

Note that the above code does not necessarily represent the actual implementation. It represents ideas that guided the code design.

Lambda function

The name "lambda function" is borrowed from functional programming where it describes an anonymous function (Hudak, 1989), in other words, a function without an identifier[5] associated with it. It is often used, however, in conjunction with list *iterators*. The iterator fetches elements from the list, whilst the lambda function does elementary operations on the fetched elements. It is in *this* sense that the term "lambda function" fits with the mathematical functions described in this paper.

By separating element-fetching and element operations, lambda functions can be constructed as scalar functions easily represented by a graph of operators and operands (previously discussed in Section 2.1).

[5] An identifier in programming is the variable name. If a function has not been declared as `def myfunction ...` or something similar, it could be viewed as a lambda function

Løvfall (2008) created a system called RGrad (also using Ruby) to construct lambda functions, using a syntax similar to Ruby's own. Multidimensional algebraic objects that appear in the lambda function are broadcasted to the same rank, then operated upon element-by-element by the lambda function which is specified in {}-brackets. Below, the expression

$$a_j = \sum_{i=1}^{c} \frac{l_{ij} n_i}{m_i} \tag{2.18}$$

has been implemented in RGrad ($a =$`llm`, $l =$`lij`, $n =$`n` and $m =$`lm`):

```
llm = RGrad::expr(lij,n[nc,nil],lm[nc,nil]){
      |_lij,_mi,_lmi| _lij*_mi/_lmi
}.sum!(nil,nc)
```

The algebraic object that is returned by the lambda function can be subject to further structure-altering operations, in this case it is summed. Separating the element-by-element operations has its advantages, seeing as differentiation here affects only the lambda function: e.g. the derivative of a sum is trivially the sum of the derivatives. Differentiation of structure-altering operators is discussed in Section 3.5.

In this paper, though, another solution was opted for: Instead of lambda functions, the lambda (scalar) operators appear alongside their structure-altering counterparts, this leads to a less bloated syntax. The implementation of the above expression using the MDO class would look like:

```
llm = (lij*mi/lmi).sum
```

In this example, the default specifications for broadcasting and summation would happen to work. Specifying them explicitly would also work:

```
llm = (lij*mi.bc(nil, nc)/lmi.bc(nil, nc)).sum(0)
```

Note that exactly what each bit of code *does* is not important at this point. The code examples are here only to show that a different approach was taken, and that the element-by-element operations are not separated into {}-brackets.

Folding and unfolding of MDOs

A much used operator in thermodynamic modelling is summation. Whether it is partial molar quantities, compositions, energy contributions or another vectorial quantity, summation plays an important role. If the summed quantity was organized in a vector of unspecified length, as would be natural for say, component mole numbers, then the summation operator takes the vector and produces a single number: The rank of the MDO is reduced by one.

In functional programming an operation that takes an array and produces a single number is called *folding*. Folding uses an operator between elements: For example a sum is a fold with the addition operator. The commas of a list `[1, 2,`

`3, 4]` is replaced by `+: 1 + 2 + 3 + 4 = 10`. If the same list is folded with `*`, the result would be `1*2*3*4 = 24` which is equal to the *factorial* of four. Meijer, Fokkinga, and Paterson (1991) generalized rank reducing operations as *catamorphisms*. Using the paper's notation, if a list type is denoted $A*$, and its element a type B, then a catamorphism is a function $\in A* \rightarrow B$. In words: A catamorphism takes a list and produces something of the same type as the elements in the list. While "catamorphism" is the word used in category theory, "fold" is the functional programming analogy.

The opposite of folding is sometimes called *unfolding*. It entails the production of a list from elements or a *seed*[6]. Some sort of production rule has to be specified that takes the seed and generates the resulting list elements. Meijer et al. (1991) generalized this concept and named it *anamorphism* as the dual to catamorphism. The most common example of this is the `zip` function. It takes two lists and makes a list where the elements of the input are listed in pairs: `zip([a, b, c], [1, 2, 3]) = [[a, 1], [b, 2], [c, 3]]`.

In the context of this work *broadcasting*, adding MDO dimensions and differentiation are examples of anamorphisms because the rank of an MDO is increased. Summation can be viewed as catamorphism, as well as indexing which picks out one specific element in a list, reducing the rank by one. Using generalized rank-increasing and rank reducing formalisms as employed by Meijer et al. (1991), it might have been possible to implement general fold and unfold operators, but that was not prioritized in the design of the `Expression` class. Implementing general folds and unfolds would have granted the user more flexibility, however, it was decided that this benefit would not be substantial enough to outweigh the cost of time that would be spent implementing it.

2.4 The gradient

The "gradient" in this paper is taken to mean the derivatives of a function with respect to all or a set of its free variables. Taking the gradient of a scalar function yields a vector

$$\nabla f(x_1, x_2, ...x_n) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n}\right] \tag{2.19}$$

However, if f itself was a vector valued function, then each component function f_i must be differentiated with respect to each free variable x_j. The result is a matrix, often called the Jacobian:

$$\nabla\left[f_1,\, f_2,\, ...,\, f_m\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_2} \\ \vdots & & \ddots & \vdots \\ \frac{\partial f_1}{\partial x_n} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{2.20}$$

[6]Seed is a value used to determine subsequent values in a sequence

How about if f was a matrix function? Each component function f_{ij} must be differentiated with respect to each differentiation variable x_k. The whole gradient can be collected in a three dimensional MDO with elements $\partial f_{ij}/\partial x_k$:

$$\nabla f = \left[\frac{\partial f_{ij}}{\partial x_k}\right] \tag{2.21}$$

Each time the gradient is taken, a new dimension is added to the MDO. This is the main reason for introducing multidimensional algebraic objects in the first place: To find the gradient to arbitrary order (taking the gradient of a function as many times as one pleases), arbitrary rank MDOs are needed.

Introducing the Kronecker delta

Above it was mentioned that the gradient is the derivative with respect to the free variables. In thermodynamics a set of the free variables may look like $\mathbf{n} = [n_1, n_2, ..., n_C]$. Say a function c took the form:

$$c = \sum_{i=0}^{c}\sum_{j=0}^{c} n_i n_j \gamma_{ij} \tag{2.22}$$

then, assuming that γ_{ij} is not dependent on $\mathbf{n}$.

$$\frac{\partial c}{\partial n_k} = \sum_{i=0}^{c}\sum_{j=0}^{c} \frac{\partial}{\partial n_k}\left(n_i n_j\right)\gamma_{ij} \tag{2.23}$$

which by using the product rule expands to

$$\frac{\partial c}{\partial n_k} = \sum_{i=0}^{c}\sum_{j=0}^{c} \left(n_j \frac{\partial n_i}{\partial n_k} + n_i \frac{\partial n_j}{\partial n_k}\right)\gamma_{ij} \tag{2.24}$$

If $\mathbf{n}$ is a free variable vector, then the elements do not depend on each other, and

$$\frac{\partial n_i}{\partial n_k} = \delta_{ik} \tag{2.25}$$

On the right hand side of Equation (2.25) δ_{ik} represents the Kronecker delta. It's definition is

$$\delta_{ij} = \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases} \tag{2.26}$$

It is used here because a variable n_i differentiated with respect to itself is 1 but its derivative with respect to any other variable is 0. The final expression for the derivative of c with respect to n_k becomes

$$\frac{\partial c}{\partial n_k} = \sum_{i=0}^{c}\sum_{j=0}^{c} \left(n_j \delta_{ik} + n_i \delta_{jk}\right)\gamma_{ij} \tag{2.27}$$

It is evident that the Kronecker delta will be a necessary component in a program to differentiate functions with respect to vector elements.

What if one wants to differentiate with respect to higher order objects, like a 2-d MDO (matrix)? Say, the derivative with respect to the parameters γ was needed:

$$\frac{\partial c}{\partial \gamma_{kl}} = \sum_{i=0}^{c} \sum_{j=0}^{c} (n_i n_j) \frac{\partial \gamma_{ij}}{\partial \gamma_{kl}} \tag{2.28}$$

Again assuming that the elements γ are independent of each other.

$$\frac{\partial \gamma_{ij}}{\partial \gamma_{kl}} = \begin{cases} 1, & i = k, j = l \\ 0, & \text{otherwise} \end{cases} \tag{2.29}$$

Which can be expressed using the Kronecker delta.

$$\frac{\partial \gamma_{ij}}{\partial \gamma_{kl}} = \delta_{ik}\delta_{jl} \tag{2.30}$$

Two Kronecker deltas multiplied together appear because of the rank 2 differentiation variable. It seems that the Kronecker delta exceeds its intended usefulness as the derivative of a vector with respect to itself: It can also be used to derivatives with respect to higher order MDOs.

$$\frac{\partial a_{i_1 i_2 \ldots i_r}}{\partial a_{j_1 j_2 \ldots j_r}} = \delta_{i_1 j_1}\delta_{i_2 j_2} \ldots \delta_{i_r j_r} \tag{2.31}$$

The symbol r denotes the rank of the differentiation variable a.

2.5 Thermodynamic modelling of phase equilibrium

The classical thermodynamic problem of multicomponent phase equilibrium is used to test the gradient calculation software. The problem is stated as follows:

$$\min_{V^v, V^l, \mathbf{n}^v, \mathbf{n}^l} (A^v + A^l)_T \tag{2.32}$$

The two phases are denoted as v and l. By introducing more equations, namely the conservation of volume and mole number

$$V^v + V^l = V_{\text{tot}} \tag{2.33}$$

$$\mathbf{n}^v + \mathbf{n}^l = \mathbf{n}_{\text{tot}} \tag{2.34}$$

$V^l, \mathbf{n}^l$ can be expressed as functions of $V^v, \mathbf{n}^v$, cutting the number of free variables in half. Equation (2.32) can now be written as

$$\frac{\partial (A^v + A^l)}{\partial \mathbf{x}^v} = \mathbf{0} \tag{2.35}$$

Where $\mathbf{x} = [V, n_1, n_2, ..., n_C]^\top$. Rearranging yields

$$\frac{\partial A^v}{\partial \mathbf{x}^v} = -\frac{\partial A^l}{\partial \mathbf{x}^v} \tag{2.36}$$

From differentiation of Equations (2.33) and (2.34), it follows that $\partial \mathbf{x}^v = -\partial \mathbf{x}^l$.

$$\frac{\partial A^v}{\partial \mathbf{x}^v} = \frac{\partial A^l}{\partial \mathbf{x}^l} \tag{2.37}$$

Equation (2.37) is often written as

$$\begin{aligned} p^v &= p^l \\ \mu_i^v &= \mu_i^l, \qquad i = 1, 2, ..., C \end{aligned} \tag{2.38}$$

Equations (2.37) and (2.38) are related by Equation (2.39)[7].

$$\begin{aligned} p &= -\left(\frac{\partial A}{\partial V}\right)_{T,\mathbf{n}} \\ \mu_i &= \left(\frac{\partial A}{\partial n_i}\right)_{T,V,n_j \neq n_i} \end{aligned} \tag{2.39}$$

Pressure, p, and chemical potential, μ_i, can be thought of as *functionals* of A. Just as a function takes on different values depending on the values of the free variables, the functionals p and μ_i take on a different function form depending on the model for A. They can therefore be applied both for the alpha phase and the beta phase for any model for A. Combining $-p$ and μ_i into a vector valued functional $\mathbf{y} = [-p, \mu_1, \mu_2, ..., \mu_C]^\top$, Equation (2.37) can be written as

$$\mathbf{y}^v(\mathbf{x}^v) = \mathbf{y}^l(\mathbf{x}^l) \tag{2.40}$$

What remains is to find an expression for $A(T, V, \mathbf{n})$. In this particular application, the Redlich-Kwong equation of state is used to derive an expression for the function $A(T, V, \mathbf{n})$ such that Equation (2.39) becomes calculable.

The Redlich-Kwong Equation of State

The Redlich and Kwong (1949) equation of state (RK-EOS) is:

$$p^{RK} = \frac{NRT}{V - b} - \frac{a}{T^{1/2}V(V + b)} \tag{2.41}$$

For a mixture of chemical components, the parameters b and a are defined as

$$b = \sum_i n_i b_i \tag{2.42}$$

[7] The negative pressure has been changed to positive pressure in Equation (2.38) by multiplying both sides of the equation by -1

$$a = \sum_i \sum_j n_i n_j a_{ij} \tag{2.43}$$

where

$$a_{ij} = (a_i a_j)^{1/2}(1 - k_{ij}) \tag{2.44}$$

The geometric mixing rule in Equation (2.44) is the most commonly employed mixing rule for cubic equations of state, such as the RK-EOS[8] (Dimian, Bildea, and Kiss, 2003).

Finally, the component specific parameters a_i and b_i can be found from critical point data through:

$$b_i = 0.0867 RT_{c,i}/p_{c,i} \tag{2.45}$$

$$a_i = 0.4278 R^2 T_{c,i}^{2.5}/p_{c,i} \tag{2.46}$$

These equations are derived from the properties of the mechanical critical point as the point of inflection on the p, V-isotherm. This condition yields two equations:

$$\left(\frac{\partial p}{\partial V}\right)_{T=T_c} = 0 \tag{2.47}$$

$$\left(\frac{\partial^2 p}{\partial V^2}\right)_{T=T_c} = 0 \tag{2.48}$$

By inserting the RK-EOS, Equation (2.41), into Equations (2.47) and (2.48) and evaluating at the known critical condition (p_c, T_c), two equations with two unknowns a and b can be solved to give Equations (2.45) and (2.46). In other words: The parameters a_i and b_i only depend on $T_{c,i}$ and $p_{c,i}$.

Helmholtz energy

As a starting point, Helmholtz energy, being a state function, can be written as

$$A(T, V, \mathbf{n}) = A^{\text{ig}}(T, V, \mathbf{n}) + A^{\text{res}}(T, V, \mathbf{n}) \tag{2.49}$$

Ideal gas: Looking only at the ideal part, at constant temperature, Helmholtz energy can be expressed as (Haug-Warberg, 2006):

$$A^{\text{ig}}(T, V, \mathbf{n}) = -p^{\text{ig}} V + \sum_{i=1}^{c} \mu_i^{\text{ig}} n_i \tag{2.50}$$

The expression for p^{ig} is simply NRT/V. The ideal gas component chemical potential, however, has a slightly more complicated expression. Equation 2.51 is adapted from Smith (2004).

$$\mu_i^{\text{ig}} = \mu_i^0(T, p_0) + RT \ln\left(\frac{n_i RT}{V p_0}\right) \tag{2.51}$$

[8]The binary interaction parameter k_{ij} is interesting to include, because it can be represented by a rank 2 MDO, as opposed to vectors like $\mathbf{n}$ which are one-dimensional

The pure component chemical potential $\mu_i^0(T, p_0)$ is still an unknown function.

For a pure component system, $\mu_i = G$, and assuming p_0 to be the reference pressure we can write:

$$\mu_i^0(T, p_0) = h_i^0(T, p_0) - T s_i^0(T, p_0) \tag{2.52}$$

Taking the reference state to be elements at standard temperature and pressure T_0, p_0, the following expressions can be used for enthalpy and entropy:

$$h_i^0(T, p_0) = \Delta_f h_i^0(T_0, p_0) + \int_{T_0}^{T} c_{p,i}^0(T) dT \tag{2.53}$$

and

$$s_i^0(T, p_0) = s_i^0(T_0, p_0) + \int_{T_0}^{T} \frac{c_{p,i}^0(T)}{T} dT \tag{2.54}$$

The final expression becomes

$$\mu_i^0(T, p_0) = \Delta_f h_i^0(T_0, p_0) + \int_{T_0}^{T} c_{p,i}^0(T) dT - T \left[s_i^0(T_0, p_0) + \int_{T_0}^{T} \frac{c_{p,i}^0(T)}{T} dT \right] \tag{2.55}$$

The standard enthalpy of formation $\Delta_f h_i^0(T_0, p_0)$ as well as the standard entropy $s_i^0(T_0, p_0)$ can be found in thermodynamic tables.

The final requirement is an expression for $c_{p,i}^0$. Parameters for the following polynomial fit were adapted from Berkeley Gas Research Institute (1999)[9]:

$$c_{p,i}^0(T) = A_i + B_i T + C_i T^2 + D_i T^3 + E_i T^4 \tag{2.56}$$

Residual: An expression for the residual Helmholtz energy can be found from

$$A^{\text{res}}(T, V, \mathbf{n}) = A^{RK}(T, V, \mathbf{n}) - A^{ig}(T, V, \mathbf{n}) \tag{2.57}$$

$A^{RK}(T, V, \mathbf{n})$ is Helmholtz energy calculated using the RK-EOS. By recognizing that $A^{ig} = A^{RK}$ at $V \to \infty$, Equation (2.57) can be rewritten. Keeping T and $\mathbf{n}$ constant, and choosing $V = \infty$ as reference state yields

$$A^{\text{res}}(T, V, \mathbf{n}) = \int_{V}^{\infty} -p^{ig} dV - \int_{\infty}^{V} p^{RK} dV \tag{2.58}$$

Flipping the limits of the first integral and omitting the unchanged T and $\mathbf{n}$ yields

$$A^{\text{res}} = \int_{\infty}^{V} \left[p^{ig}(V) - p^{RK}(V) \right] dV \tag{2.59}$$

[9]The parameter values were originally formulated for the unit-less polynomial $c_{p,i}^0/R = a_i + b_i T + c_i T^2 + d_i T^3 + e_i T^4$. By multiplying each parameter by R, the unit-specific parameters A_i, B_i, etc. could be found

The expressions for $p^{ig}(V)$ and $p^{RK}(V)$ can be inserted into Equation (2.59) so that the integral can be evaluated.

$$A^{\text{res}} = \int_{\infty}^{V} \left[\frac{NRT}{V} - \frac{NRT}{V-b} + \frac{a}{T^{1/2}V(V+b)} \right] dV \tag{2.60}$$

The integral in Equation (2.60) evaluates to

$$A^{\text{res}} = NRT \ln\left(\frac{V}{V-b}\right) + \frac{a}{bT^{1/2}} \ln\left(\frac{V}{V+b}\right) \tag{2.61}$$

Iteration scheme

To recap the notation used previously in this section:

$$\mathbf{x} = \begin{bmatrix} V \\ n_1 \\ n_2 \\ \vdots \\ n_C \end{bmatrix}, \qquad \mathbf{y} = \frac{\partial A}{\partial \mathbf{x}} = \begin{bmatrix} -p \\ \mu_1 \\ \mu_2 \\ \vdots \\ \mu_C \end{bmatrix} \tag{2.62}$$

The equilibrium condition from Equation (2.40) is

$$\mathbf{y}^v(\mathbf{x}^v) = \mathbf{y}^l(\mathbf{x}^l) \tag{2.63}$$

The left and right hand side of Equation (2.63) can be Taylor-expanded around arbitrary points $\mathbf{x}_k^v$ and $\mathbf{x}_k^l$ respectively.

$$\mathbf{y}^v(\mathbf{x}_k^v) + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^v \Bigg|_{\mathbf{x}_k^v} \Delta\mathbf{x}_k^v = \mathbf{y}^l(\mathbf{x}_k^l) + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^l \Bigg|_{\mathbf{x}_k^l} \Delta\mathbf{x}_k^l \tag{2.64}$$

The conservation of volume and mole number reads: $\mathbf{x}^v + \mathbf{x}^l = \mathbf{x}_0$, where $\mathbf{x}_0$ is the constant total volume and mole numbers. Differentiating the constraint yields the useful relationship: $\Delta\mathbf{x}^v = -\Delta\mathbf{x}^l$. Inserting this into Equation (2.64) gives Equation (2.65).

$$\mathbf{y}^v(\mathbf{x}_k^v) + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^v \Bigg|_{\mathbf{x}_k^v} \Delta\mathbf{x}_k^v = \mathbf{y}^l(\mathbf{x}_k^l) - \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^l \Bigg|_{\mathbf{x}_k^l} \Delta\mathbf{x}_k^v \tag{2.65}$$

The last equation can be rearranged to yield an explicit expression for the iteration variable increment:

$$\Delta\mathbf{x}_k^v = \left[\left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^v \Bigg|_{\mathbf{x}_k^v} + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top}\right)^l \Bigg|_{\mathbf{x}_k^l} \right]^{-1} \left(\mathbf{y}^l(\mathbf{x}_k^l) - \mathbf{y}^v(\mathbf{x}_k^v)\right) \tag{2.66}$$

Renaming parts of the above equation according to

$$H(\mathbf{x}^v, \mathbf{x}^l) = \left[\left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^v \Bigg|_{\mathbf{x}_k^v} + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^l \Bigg|_{\mathbf{x}_k^l} \right] \tag{2.67}$$

$$\mathbf{g}(\mathbf{x}^v, \mathbf{x}^l) = \left(\mathbf{y}^l(\mathbf{x}_k^l) - \mathbf{y}^v(\mathbf{x}_k^v) \right) \tag{2.68}$$

yields:

$$\Delta \mathbf{x}_k^v = H^{-1}(\mathbf{x}^v, \mathbf{x}^l)\mathbf{g}(\mathbf{x}^v, \mathbf{x}^l) \tag{2.69}$$

$H^{-1}(\mathbf{x}^v, \mathbf{x}^l)$ can be recognized as the inverse Hessian of total Helmholtz energy[10]. The second factor, $\mathbf{g}(\mathbf{x}^v, \mathbf{x}^l)$, is the first derivative of total Helmholtz energy. In order to calculate the next $\Delta \mathbf{x}^v$, the free variables $\mathbf{x}^v$, $\mathbf{x}^l$ must be updated according to Equations (2.70).

$$\mathbf{x}_{k+1}^v = \mathbf{x}_k^v + \Delta \mathbf{x}_k^v, \qquad \mathbf{x}_{k+1}^l = \mathbf{x}_k^l - \Delta \mathbf{x}_k^v \tag{2.70}$$

When the norm of the residual reaches below a certain value,

$$\|\mathbf{g}(\mathbf{x}_k^v, \mathbf{x}_k^l)\| < \epsilon \tag{2.71}$$

iteration ends and the solution is taken to be $\mathbf{x}_k^v, \mathbf{x}_k^l$.

As can be seen from Equation (2.69) this is really just Newton's method applied to the function $\mathbf{g}(\mathbf{x}^v, \mathbf{x}^l) = \mathbf{0}$. According to Kelley (2003), Newton's method has quadratic order of convergence which means that the number of significant figures in the result (approximated solution) roughly doubles with each iteration.

In the application of the MDO computer algebra system (Chapter 4.2), the functions $\mathbf{y}(\mathbf{x})$ and $d\mathbf{y}/d\mathbf{x}^\top$ are expressed analytically and such that they are evaluable for both phases v and l.

[10] In the Newton scheme $H^{-1}(\mathbf{x}^v, \mathbf{x}^l)$ takes the role of inverse Jacobian

Chapter 3

The MDO class design

The `MDO` class was written in the Ruby programming language. All programming examples are therefore given using a Ruby-like pseudo-code. The code as well as supporting example files can be found in Appendix A. The reader is assumed to have some knowledge of object-oriented programming, algorithms and data-structures.

This chapter provides insight into how the `MDO` class works: How to build functions using `MDO` objects and how to evaluate and differentiate these functions. Mathematical expressions using the `MDO` class are stored in a graph structure where each node contains an operator. The first part of the chapter covers how to build basic `MDO` graphs using the scalar operators (+, -, * ... etc.). More advanced operators that affect the dimensionality of an `MDO`, like summation and indexing, are discussed towards the end of the chapter.

This chapter does not show how to *use* the `MDO` class. To see the `MDO` syntax in action, refer to Section 4.2.

3.1 Concept

An `MDO` object is not a value, but an MDO expression (or function): The data that makes up an `MDO` object is a recipe for how to calculate values, not the values themselves. One cannot differentiate a value, an expression on the other hand can be differentiated, evaluated or used to construct more complex expressions. *Higher order programming* is a name often used about functions which take functions as arguments and produce new functions (Dominus, 2005). In this sense, differentiation can be thought of as a *higher order function*.

Two free variables, `a` and `b` can be initialized as `MDO` objects

```
a = MDO::new
b = MDO::new
```

The current memory layout of the computer program is visualised in Figure 3.1.

Figure 3.1: A graph representation of two free variable MDOs, `a` and `b`, the square brackets mean that they don't have dimensions (they are scalars).

Writing

```
c = a*b
```

does *not* give `c` the value of of `a` times `b` (`a` and `b` haven't even been given values yet). Instead it stores the operator `*` and references to `a` and `b`, so that *when* `c` is evaluated it can both fetch the values of `a` and `b`, *and* know what to do with them (multiply). The structure of `c` has been visualised in Figure 3.2

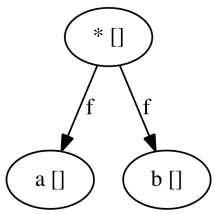

Figure 3.2: The graph structure of `c=a*b`. The variable name `c` refers to the top node. Empty square brackets indicate no dimensions which means that the nodes represent scalar expressions. `f` stands for "factor".

The expression can be augmented further:

```
d = c+a
```

The structure of the `MDO` object `d` can be seen in Figure 3.3.

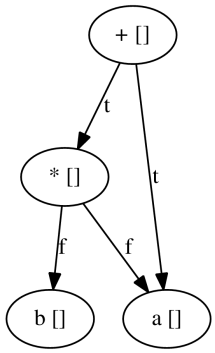

Figure 3.3: The graph structure of `d=c+a` where `c=a*b`. The variable name `d` refers to the top node. Empty square brackets indicate no dimensions which means that the nodes represent scalar expressions. `f` stands for "factor" and `t` stands for "term"

An `MDO` expression does not have to be constructed one operator at a time, writing

```
d = a*b+a
```

also results in `d` having the graph structure in Figure 3.3. The terms; `MDO` object, node, graph, expression and `MDO` function are all used to describe `MDO` objects. They are all valid, but different, interpretations of what an `MDO` object represents.

3.2 Class members

An object of the `MDO` class has several accessible *members*[1]: `operator`, `dim`, `dep`, `val` and `label`. Each of them are explained below.

`operator`

This member variable contains the outer operator of the expression represented by an `MDO` object. For example: Writing

```
c=a/b
```

gives `c` the / operator. Each of the different operators has a sub-class within the `MDO` class:

[1]The variables inside an object are called *members*. Different objects of the same class may have different values for their members.

```
class MDO
      ...
      class Div
            ...
      end
      ...
      class Mult
            ...
      end
      ...
      class Add # etc.
            ...
      end
      ...
end
```

In the above example (`c=a/b`), what really happens is that a new `MDO`, `c` is created and its `operator` is assigned an object of the `Div` class:

```
c.operator = Div::new(...)
```[2]

All the information about how the division operator works is contained in the `MDO::Div` class. All nodes but independent variables and constants are given an operator this way. Independent variables and constants are not assigned operators, so they have `operator = nil`[3].

Operator classes have been divided into two distinct categories: Lambda operators and structural operators. Lambda operators are scalar operators such as `+`, `-`, `*`, `/`, `ln`, `exp`. Structural operators include summation, indexing, adding an MDO dimension and broadcasting. When an `MDO` object is evaluated, lambda operators determine the numbers, while structure operators determine where numbers are placed in the MDO.

Defining each operator in its own class has the benefit of providing easy extensibility: When a new operator is desired it needs only to be added as a new class along with the already existing operators. All the functionality specific to that operator is collected in one place.

`dim`

An array of dimension sizes for the MDO object. For example an MDO representation of a 2×3 matrix would have `dim = [2, 3]`, while a scalar function would have an empty array, i.e. `dim = []`. The rank of the MDO is the length of the `dim` array.

[2] Wherever a function argument is not of interest to the present discussion, ellipses are used. For example, the `eval` function may take several arguments, but when they are not of importance, the function call will appear in the text as `eval(...)`.

[3] In Ruby, `nil` is a value representing "nothing", or "empty"

Instead of constant numbers, the `dim` array may also contain *dimension* variables. For example, in the definition $\mathbf{n} = [n_1, n_2, ..., n_C]$, the letter C could be said to denote a dimension variable. Just like the expression `c=a*b` doesn't store the value of `a*b`, the dimensions sizes don't have to be values either. The value of the dimension variable would have to be specified upon evaluation, just like a free variable. Otherwise the evaluation method would not know how many MDO elements it needs to calculate. Section 3.7 provides more information on how to create and use dimension variables.

`dep`

An array of child `MDO` object references. The elements of this array are not necessarily the dependent variables, but references to sub-expressions, for example: An `MDO` object `c`, defined as `c = a-(b+a)`, contains the `-` (subtraction) operator, its `dep` array has two elements, a reference to `a` and a reference to `a+b`. The `MDO` object `a+b` contains the `+` operator and references to `a` and `b` in the `dep` array: The references to `a` and `b` represent the operands of the `+` operator. Constants and independent variables have no dependent nodes which means they have `dep = []`.

`val`

One or more parameter values (pure numbers) used to fully specify an operator. For example a constant node with `operator = nil` and `dep = []` needs a place to store the numeric value. So the value is stored in `val`. Other operators that use `val` include broadcasting, indexing, powers and summation. Because of Ruby's flexibility, the contents of `val` could have been stored elsewhere, for example in `dep`, thereby eliminating the need for an extra variable. It was however prioritized to make `dep` contain only `MDO` references, whilst constant parameters are stored in `val`.

`label`

A string used to identify the node when exported to other code formats. Can also be useful for distinguishing between independent variables in a graphical representation of the function graph.

3.3 Graph traversal

Traversing a graph means visiting all its nodes in a particular order. It distinguishes itself from *tree* traversal where each node is visited exactly once: In graph traversal, each edge (arrow/relation) is visited only once (Valiente, 2002).

Traversing an expression graph can be done by calling a method on the root object, which calls the same method on its child objects (who again calls the method on their children). It is similar to *recursion*, meaning a function calling itself. What separates traversal from recursion is that the method calls itself, but on another

object. There are two ways of traversing an `MDO`: Evaluation and differentiation. The evaluation method has been called `eval`[4]. Differentiation can be done by calling the `grad` method: It not only visits the nodes of the graph, but returns a new graph representing the derivative expression.

Scalar evaluation

A small program which utilized the evaluation method may look like:

```
# initialize two independent variables
a = MDO::new #
b = MDO::new #

c = a*b+a
puts c.eval({a=>4.0, b=>2.0}) # => 12.0
puts c.eval({a=>2.0, b=>3.0}) # => 8.0
```

[5]

The state of the evaluation ($a = 4, b = 2$ in the first `eval` call) is specified through a *hash table*[6] which associates `a` with the value `4.0` and `b` with the value `2.0` (The `=>` symbol is used in Ruby to associate hash keys with hash values).

The evaluation of the `c` node requires evaluation of its child nodes. The child nodes in turn require the evaluation of their child nodes and so on until the leaf nodes are met. Figure 3.4 shows how a call to `eval` traverses the graph. When the leaf nodes are reached, the `eval` function finds the specified value of the independent variable in the hash table given as argument. Since leaf node `MDO`s have no children, no further `eval` calls are made, and the whole expression has been evaluated.

Scalar function differentiation

Differentiation distinguishes itself from evaluation in that it does not return numeric values, but a new `MDO` object. Differentiation can be called through the `grad` method as the following example illustrates:

---

[4] Not to be confused with Ruby's own `eval` function. Ruby's eval function executes a string of Ruby code from within a script.

[5] `c.eval` means that the *method* `eval` is called from the `MDO` object `c`. An object *method* works with the input (here, {`a=>4.0, b=>2.0`}) and the member values of the object. Looking at the expression for `c`, it has the `operator.class = Add`, `eval` will end up adding some numbers in this case. The method `eval` will call `eval` on `c`'s children. Since these children again will call `eval` on their children, `eval` is said to traverse the graph.

[6] A hash table is a data structure that provides an easy way of looking up values from associated keys (here, the independent variable MDOs `a` and `b` are the keys). In Ruby, a hash table is specified with {}-braces, and relations are specified with an arrow `a=>b`

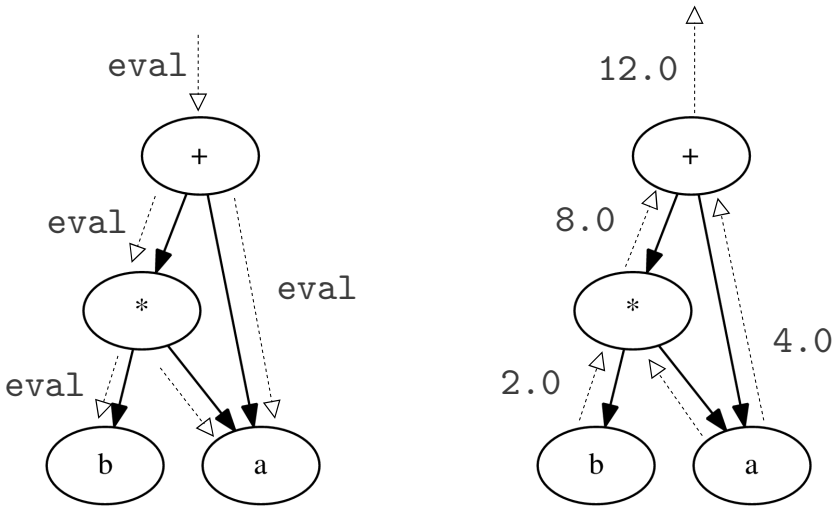

Figure 3.4: A graph representation of the function `c=a*b+a`. The dotted line arrows labelled "eval" show the `eval` method traverses the graph, while the dotted line arrows labelled with numbers shown the return values of the same calls.

```
# initialize two independent variables
a = MDO::new
b = MDO::new

c = a*b+a
d = c.grad(a)
```

Writing `c.grad(a)` really means $\partial c/\partial a$. The function `d` should now be something equivalent to `b+1`.

The way in which `grad` works is best seen in Figure 3.5. The gradient call is starts at the top node (named `c` in the above example). It has `operator.class = Add`, so it simply returns

$$\frac{\partial}{\partial a}(f+g) = \frac{\partial f}{\partial a} + \frac{\partial g}{\partial a} \tag{3.1}$$

The sub-figures 1-5 are listed in chronological order as different stages encountered when the `grad` function is called. Opened, and unreturned[7] calls are represented by downwards facing arrows pointing at red colored nodes. In the time between sub-figures 3.5-1 and 3.5-2, a new + node is created (colored blue) and the gradient call is let loose on the old + node's children. Between sub-figures 3.5-2 and 3.5-3, `a` is differentiated to the constant `1`, because `a` is the differentiation variable. Between sub-figures 3.5-3 and 3.5-4, the node with the `*` sign is differentiated by the product rule:

$$\frac{\partial}{\partial a}(ab) = a\frac{\partial b}{\partial a} + b\frac{\partial a}{\partial a} \tag{3.2}$$

Finally, between sub-figures 3.5-4 and 3.5-5, the nodes a and b differentiate to 1 and 0 respectively.

Taking the graph in Figure 3.5-5 and turning it back to a mathematical expression yields `(b*1+0*a)+1` which indeed is equivalent to `b+1` and is therefore a valid gradient expression. It is evident that a simplification algorithm would be beneficial, as multiple gradients of the same function will quickly result in increasingly large graphs. Reduction is covered in Section 3.8.

[7] When a function is called, its code is executed from the top down. At a point in time it might not have reached the end (or the `return` statement), this is what is meant by *unreturned*: A function still under execution.

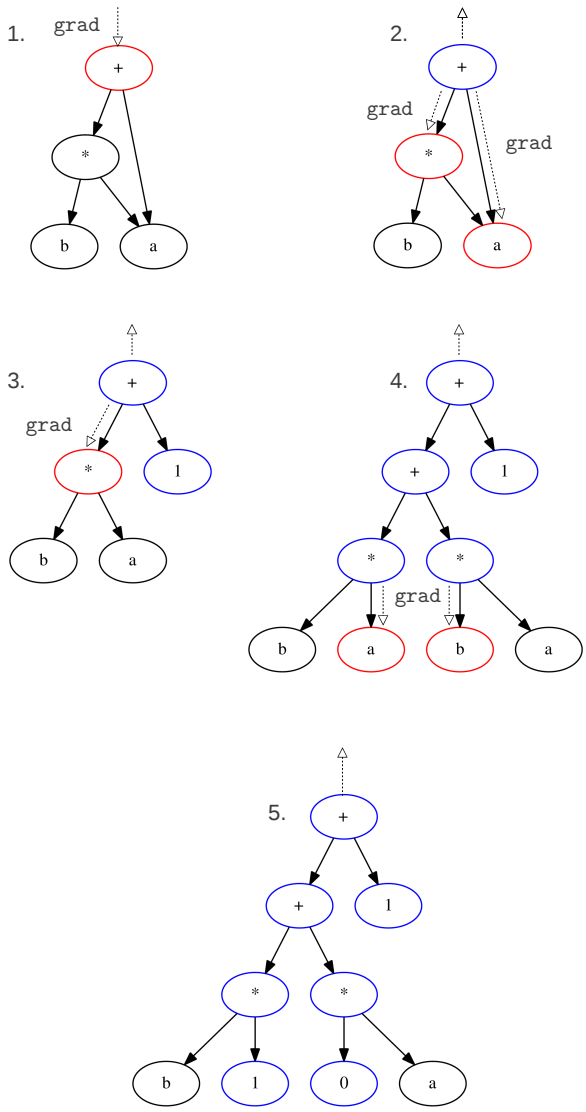

Figure 3.5: The development of the function graph `a*b+a` during a recursive *gradient* function call ($\partial/\partial a$). When the gradient is called on a node (marked red), a differentiation rule specific for the operator at that node creates new graph nodes (shown in blue), and calls for differentiation of child nodes. 1. The + node is to be differentiated. 2. The resulting graph after the + node is differentiated. 3. Result after the independent variable `a` was differentiated. 4. The multiplication node has been differentiated using the product rule. 5. The whole graph has finally been differentiated. Now the graph represents the expression `1*b+0*a+1`.

The following sections go more into more depth with regard to the operator types; what they produce when evaluated, how they are differentiated, and how they utilize the accessible members of the `MDO` class.

3.4 Lambda operators

This section describes each of the lambda operators in detail. As explained in Section 3.2, the lambda operators are scalar operators. All the operators `+ - * /` are *overloaded*[8]. This was an easy choice considering the benefit of the Ruby parser handling the order of operations automatically. The syntax is also very clean. Implementing the equation $a = b + cb$ can by done as

```
a=b+c*b
```

Because the operators are properly handled by the programming language, multiplication is automatically recognized as the innermost operation, followed by addition. The above example would have looked different if other solutions were opted for. Using methods for example, it could look like:

```
a=b.add(c.mult(b))
```

or, circumventing the Ruby parser, the `MDO` class could parse strings to create expressions. That could look something like:

```
a=MDO::parse('b+c*b') # not implemented!
```

The two latter alternatives give a more lengthy syntax than operator overloading.

Addition

An `MDO` object with `operator.class = Add` has two `MDO` references in the `dep` array, each representing a term. The `Add` operator takes two arguments, call them d_0 and d_1. When `eval` is called on an `MDO` object with `operator.class = Add` it returns the value of

$$v = d_0 + d_1 \tag{3.3}$$

It returns the function

$$f(x, ...) = \frac{\partial d_0}{\partial x} + \frac{\partial d_1}{\partial x} \tag{3.4}$$

when differentiated with respect to x.

[8] Overloading an operator is similar to defining a method. However, they look different when called. In Ruby, if an object method was defined as `def mult(b)`, then an operator to do the same could be defined as `def *(b) return mult(b)`. Now, writing `a.mult(b)` would be equivalent to writing `a*b`

Subtraction

The subtraction operator is handled in much the same way as the addition operator. The only difference is that distinction must be made between the *minuend* and the *subtrahend*. The *difference* = *minuend* - *subtrahend*. In practice this means that the first element of `dep` becomes the *minuend* and the second element of `dep` becomes the *subtrahend*. Upon evaluation a node with `operator.class = Sub` returns

$$v = d_0 - d_1 \tag{3.5}$$

It returns the function

$$f(x, ...) = \frac{\partial d_0}{\partial x} - \frac{\partial d_1}{\partial x} \tag{3.6}$$

when differentiated with respect to x.

Multiplication

An `MDO` node with `operator.class = Mult` has two child nodes, each representing a factor. When evaluation is called it returns the product of the two child nodes:

$$v = d_0 d_1 \tag{3.7}$$

When a multiplication node is differentiated with respect to an MDO x, the function returned is according to the product rule:

$$f(x, ...) = d_1 \frac{\partial d_0}{\partial x} + d_0 \frac{\partial d_1}{\partial x} \tag{3.8}$$

Division

Division nodes have `operator.class = Div`. Like the subtraction operator, a distinction must be made between the node children. The first child `dep[0]` contains a reference to the `MDO` for the *numerator*. The second child `dep[1]` contains a reference to the *denominator* `MDO`. When evaluated, a division node returns the value of

$$v = d_0/d_1 \tag{3.9}$$

A division node returns the function

$$f(x, ...) = \frac{\partial d_0}{\partial x}/d_1 - \frac{\partial d_1}{\partial x}\frac{d_0}{d_1^2} \tag{3.10}$$

when differentiated with respect to an MDO x.

Powers

A node with `operator.class = Pow` has two child nodes. The first, `dep[0]`, is the base, while the second `dep[1]` is the exponent. So far only constant exponents are supported, because of their ease of differentiation. The `Pow` node returns

$$v = (d_0)^{d_1} \tag{3.11}$$

when evaluated, and the function

$$f(x, ...) = d_1 \left(d_0\right)^{d_1 - 1} \frac{\partial d_0}{\partial x} \tag{3.12}$$

when differentiated with respect to x.

Natural logarithm

Unlike the `MDO` node types described above, a node with `operator.class = Ln` has only one dependent, representing the argument to the natural logarithm function. Evaluation of a node of this type yields the value

$$v = \ln\left(d_0\right) \tag{3.13}$$

when evaluated, and the function

$$f(x, ...) = \frac{\partial d_0}{\partial x} / d_0 \tag{3.14}$$

is produced when differentiated with respect to another MDO x.

Exponential

An exponential MDO node has `operator.class = Exp` and one dependent node `dep[0]` which represents the argument to the exponential function. When evaluated an exponential node returns the value of

$$v = e^{d_0} \tag{3.15}$$

And upon differentiation it returns the function

$$f(x, ...) = \frac{\partial d_0}{\partial x} e^{d_0} \tag{3.16}$$

.

Constants and free variables

A node with no type (`operator = nil`) and no dependent nodes is either a constant or a free variable. If there is a value stored in `val`, it means that the node represents a constant and `val` is used in the evaluation of the MDO. If `val = nil` then it is a free variable. During evaluation calls, the values of free variables must be specified in a hash table (discussed in Section 3.3). If a node is found to be a free variable, then the hash-table argument to the `eval` function is searched to find a value corresponding with the node being evaluated.

`MDO` objects to represent constants are created by the `MDO::const(value)` constructor[9], while free variables use the default `MDO::new` constructor.

[9] A constructor, as opposed to a method, is not called from an existing object as `a.method`, but called as a traditional function `a = constructor()`. It returns an object of the class for which it is a constructor. For example `a = MDO::const(...)` makes `a` an `MDO` object.

An example application

The following example utilizes all the concepts of the `MDO` class that have been introduced up until now. Say a function

$$z(x, y) = \ln(x + y) - \frac{\exp(y)}{x^2} \tag{3.17}$$

is to be differentiated and evaluated. Equation (3.17) differentiated is

$$\frac{\partial z(x, y)}{\partial x} = \frac{1}{x + y} - \frac{2\exp(y)}{x^3} \tag{3.18}$$

Using the `MDO` class to code this up looks like:

```
x = MDO::new # create some independent variables
y = MDO::new

z = (x+y).ln - y.exp/x**2[10]

# print the value of z at x=2 and y=3
puts z.eval({x=>2.0, y=>3.0})

dzdx = z.grad(x) # take the derivative with respect to x
```

One noticeable feature is that the operators `ln` and `exp` are written in post-fix notation. An advantage of using `x.ln` instead of `ln(x)` is that the global namespace remains uncluttered. Apart from the post-fix notation, the definition of `z` looks almost exactly the same as in Equation (3.17).

If one more line

```
dzdx.dump_graph('example') # creates a file called 'example.gv'
```

is added, it is possible to take a look at the structure of `dzdx`. The method `dump_graph` translates the `MDO` graph to the DOT language (`.dot` or `.gv` file extension). There exists a plethora of tools to convert DOT files to vector graphics, thus making them viewable. The output file from the `dump_graph` method is shown in Figure 3.6

[10] In Ruby, `**` means "to the power of"

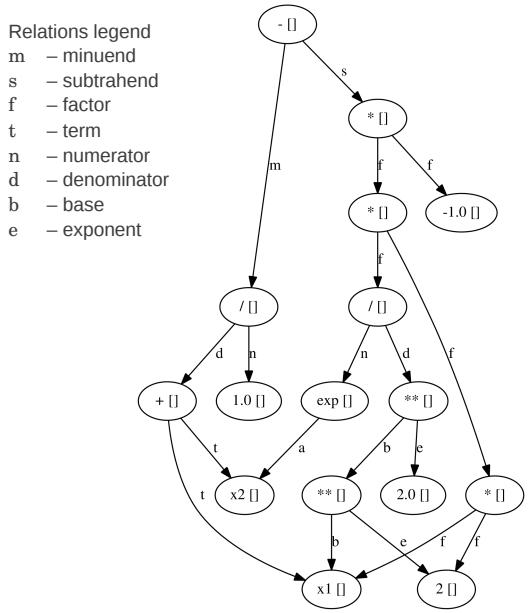

Figure 3.6: A graph representing `dzdx = z.grad(x)` where `z = (x+y).ln - y.exp/x**2`. Nodes are labelled with their operators. The empty square brackets in each label indicates the dimensions of the MDO, they are all empty because `z` is a scalar function. The free variables `x1` and `x2` are `x` and `y` respectively. By interpreting the graph it is possible to deduce its symbolic equivalent: `1.0/(x+y)-(-1.0)*(2*x*(y.exp/((x**2)**2.0)))` or $1/(x+y) + 2\exp(y)/x^4$ on simplified mathematical form.

3.5 Structure operators

Structure operators as opposed to lambda operators utilize MDO dimensions and indices. Structure operators are needed to give a non-zero rank to an MDO. These operators can be roughly categorized as either rank increasing, rank decreasing or rank conserving. The rank increasing operators include *broadcasting*, adding an MDO dimension. On the other hand, the *summation* and *indexing* operations are rank decreasing. The generalization of these kinds of operations is described in Section 2.3. Structure operators, like lambda operators, are added to an `MDO` object as additional graph nodes.

The problem of evaluating a whole MDO can be broken down into evaluating each element of an MDO. What separates the elements from one another are indices. If elements are evaluated one at a time, the scalar evaluation routine described in Section 3.3 can still be used. In order to give the user the opportunity to evaluate

the whole MDO at once, a simple wrapper has been added to the outside of the evaluation routine which finds all index combinations to evaluate and organizes the results into nested lists. Take for example a 2×2 MDO given by

$$a = \begin{bmatrix} [1 & 2] \\ [3 & 4] \end{bmatrix} \tag{3.19}$$

Instead of passing the indices as a second argument to `eval` like this

```
a00_val = a.eval({}, [0, 0])
a10_val = a.eval({}, [1, 0])
a01_val = a.eval({}, [0, 1])
a11_val = a.eval({}, [1, 1])
```

one can simply write

```
a_val = a.eval({})
p a_val # => [[1, 2], [3, 4]]
```

Both the above examples would work, the first one is clearly the more tedious one. When `eval` is called on a rank > 0 MDO and no indices are specified it automatically breaks the problem down to multiple scalar evaluations. This means that when it is time to evaluate the function's nodes, the evaluation routine always has a specified index combination to evaluate.

In the following subsection, each structure operator is discussed. The goal is to give an overview over how each operator works, and how to use it. Differentiation is not mentioned for the most part, as none of the structure operators seem to be affected by differentiation. Summation happens to be a linear operator, and therefore is unaffected by the derivative. Summation is set apart from the rest of the structural operators as it not only changes the indexing of an MDO, but does so with the help of a lambda operator (+ between each element), which happens to be linear.

The rest of the structure operators however are independent of lambda operators, and affect only the indexing of an MDO. These include indexing, permutation[11] (transposition), broadcasting and adding a dimension. Not affecting the MDO values themselves, only how they organized in the MDO makes these operators independent of differentiation. A structure operator S that takes an MDO from rank r to rank R can be loosely defined as something that changes the indexing of an MDO according to:

$$a_{i_1 i_2 \ldots i_r} \xrightarrow{S} a_{j_1 j_2 \ldots j_R} \tag{3.20}$$

The new indices $j_1 j_2 \ldots j_R$ are a function, call it s, of the old indices $i_1 i_2 \ldots i_r$:

$$S a_{i_1 i_2 \ldots i_r} = a_{s(i_1, i_2, \ldots, i_r)} \tag{3.21}$$

[11] Permutation is not discussed further in this section as it was not prioritized highly enough to fully support in the code. Neither is it used in the phase equilibrium calculations.

Indexing itself is a linear operator

$$\begin{aligned}(a+b)_i &= a_i + b_i \\ (\alpha a)_i &= \alpha a_i\end{aligned} \tag{3.22}$$

Therefore a function that only affects indices may be argued to be linear as well

$$\begin{aligned}S(a_{i_1 i_2 \ldots i_r} + b_{j_1 j_2 \ldots j_r}) &= S a_{i_1 i_2 \ldots i_r} + S b_{j_1 j_2 \ldots j_r} \\ S(\alpha a_{i_1 i_2 \ldots i_r}) &= \alpha S a_{i_1 i_2 \ldots i_r}\end{aligned} \tag{3.23}$$

Linear operators are commutative (their order can change). Therefore, the derivative ∂ of a structure operator is just the structure operator applied to the derivative:

$$\partial S a = S \partial a \tag{3.24}$$

Adding a dimension (`MDO`)

This operator is at the heart of the MDO class. To create a new dimension containing `MDO` objects, the constructor `MDO::MDO(mdo1, mdo2, ..., mdoN)` is used. The constructor returns an `MDO` object with `operator.class = MDOop`, and the arguments `mdo1, mdo2, ..., mdoN` as children (in the `dep` array). In mathematical terms, the operator does

$$a, b, c \rightarrow [a, b, c] \tag{3.25}$$

In the following example the vector function $f(x, y) = [2x,\ y+x,\ y^2]$ is constructed using the `MDO::MDO` function and evaluated at $(x = 1, y = 2)$:

```
x = MDO::new
y = MDO::new

f = MDO::MDO(2.0*x, y+x, y**2.0)
p f.eval(x=>1.0, y=>2.0) # => [2.0, 3.0, 4.0]
```

It can be used subsequently to construct MDOs of higher rank. Continuing the above code:

```
g = MDO::MDO(f, 2.0*f, 3.0*f)

p g.eval(x=>1.0, y=>2.0) # => [[2.0, 3.0, 4.0], [4.0, 6.0, 8.0],
[6.0, 9.0, 12.0]]
```

When an MDO dimension is added, the `dep` array is populated by the elements that make up the new dimension; where each element is given as an argument to `MDO::MDO`. The `dim` array in that node adds a dimension length corresponding to the number of elements. In the above example, `f` has dimension `[3]` while `g` has dimension `[3, 3]`.

Just like the lambda operators are defined as their own classes, structure operators have their own classes as well. An `MDO` object created with the function `MDO::MDO(...)` is assigned an operator of the `MDOop` class.

Earlier it was mentioned that the evaluation of an MDO of rank > 0 is done by evaluating one combination of indices at the time. When a node with `operator.class = MDOop` is evaluated it uses the first supplied index to find the corresponding child node and evaluates that (the index is used to dereference the MDO). To show how the evaluation works, take an MDO defined as

```
x = MDO::new

h = MDO::MDO(x, 2.0*x, 3.0*x)
```

It can be evaluated with the index `1`

```
p h.eval(x=>1.0, [1]) # => 2.0
```

The first and only index, `1`, calls for the evaluation of child number 1; `dep[1]`, which is `2.0*x`, producing `2.0`.

Continuing the previous example and making it a little less trivial, `y` can be defined as:

```
y = MDO::MDO(h, 10.0*h)
```

Typing

```
p y.dim
```

Reveals that `y` has dimensions `[2, 3]`. The structure of `y` can be seen in Figure 3.7. A nested list representation of `y` is

$$y(x) = \begin{bmatrix} [x & 2x & 3x] \\ [10x & 20x & 30x] \end{bmatrix} \tag{3.26}$$

Being of rank 2, `y` can be evaluated with two indices. Take `[0, 1]` as an

```
p y.eval(x=>1.0, [0, 1]) # => 2.0
```

How did it get to the right answer? The first index, `0`, was used to dereference `y`, the outermost `MDOop`-node. The second index, `1` was used to dereference the innermost `MDOop`-node, `h`.

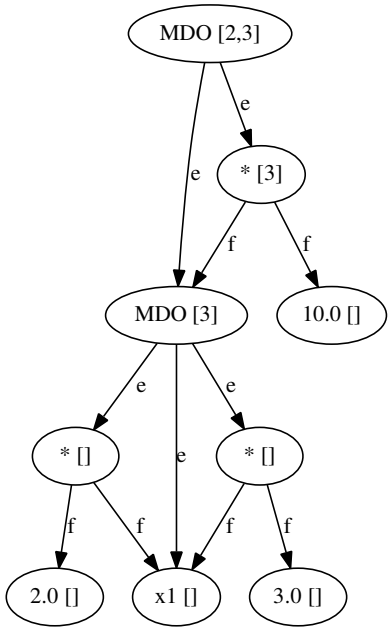

Figure 3.7: The MDO graph representation of the function $y(x) = [h(x), 10h(x)]$ where $h(x) = [x, 2x, 3x]$. Dimension of each node is shown in brackets beside the operator type. $h(x)$ is the subgraph from the node labelled "MDO [3]" and down. Arrow labels 'e': element, 'f': factor.

Summation (`sum`)

The summation operator adds all the elements in a dimension together. Visualize a dimension collapsing, leaving the sum of what was there before. The user must specify which dimension to sum, and the MDO to be summed must have at least rank 1. Calling the `sum(sum_dim)` method on an `MDO` object returns a new `MDO` object with `operator.class = Sum`. The argument `sum_dim` indicates which dimension to be summed and is stored in `val`: For example, if a rank 3 MDO with `dim = [2, 3, 4]` were to be summed once, there are three options: `val = 0` indicates that the first dimension should be summed and the resulting rank 2 MDO would have `dim = [3, 4]`. Another option is `val = 1` where the second dimension is summed and the resulting rank 2 MDO have `dim = [2, 4]`, and finally `val = 2` gives a resulting MDO with `dim = [2, 3]`.

Broadcasting (`bc`)

The theory behind broadcasting is described in detail in Sections 2.2 and 2.2. Within the MDO class it acts as an index filter, choosing the indices to be passed down to the lower rank broadcasted object. When broadcasting, one or more new

dimensions are added to the MDO. The added dimension sizes have to be specified as arguments to the broadcasting function. An argument equal to `0` or `nil` means that the MDO is not broadcasted in this dimension, and the dimension of the original MDO will be used. Take for example an MDO `a` with `dim = [2, 4]`. If this MDO is broadcasted as `b = a.bc(3, 0, 5, 0)`, `b` would have `dim = [3, 2, 5, 4]`. The old dimensions of `a` appear in order where the argument `0` was given in the broadcasting call. The zero argument can be interpreted as "do not broadcast in this direction". `a.bc(...)` returns an MDO with `operator.class = Broadcast`. The arguments `3, 0, 5, 0` are stored in `val` as an array.

The broadcasting from `a` to `b` can be shown symbolically as

$$b_{ijkl} = a_{jl} \tag{3.27}$$

Equation 3.27 is representative what happens when an MDO with `operator.class = Broadcast` is evaluated. `b` is of rank four and therefore evaluated using four indices. Upon evaluation of `b`, the second and fourth indices are passed to `a`, as the second and fourth dimensions are not broadcasted to.

Indexing (`[]`)[12]

Supplying an index to an MDO eliminates a dimension, just like the summation operator does. Where the sum adds all the elements present in that dimension, the indexing operator chooses the element corresponding to the supplied index. The indexing operator uses square brackets: `[i, d]` in the tradition of many programming languages (including Ruby). The first argument `i` is the index and the second argument is the position of the dimension on which it works (default is 0, meaning the "outer" dimension). Say `a = [[1, 2], [3, 4]]`. Indexing this as `b = a[0]` would make `b` yield `[1, 2]` when evaluated. `a[0, 1]` on the other had would yield `[1, 3]` when evaluation.

If one is not sure yet what value the index should be, one can set up an `Index` variable. For all purposes this variable is just a scalar integer MDO. This index variable must be given as the first argument in the indexing brackets, then given a value in the hash table passed to the evaluation function. In the following example, the auxiliary constructor `MDO::express` has been used to easily create a constant MDO from a nested list.

```
a = MDO::express([[1, 2], [3, 4]])

p a[0].eval() # => [1, 2]
p a[0, 0].eval() # => [1, 2]
p a[1].eval() # => [3, 4]
p a[1, 0].eval() # => [3, 4]
```

[12] The indexing operator should not be confused with the index argument supplied to the `eval` method. The results are similar, but the indexing operator adds a node to an MDO's graph structure, whereas specifying indices as an argument to the `eval` method affects only the outcome of that evaluation

In the above code, indices `0` and `1` are applied in turn to the first dimension (the default dimension, outer dimension), producing what would be called the *rows* if `a` were a matrix. To pick out what corresponds to *columns*, (the second/inner dimension), the dimension counter `1` must be supplied as the second argument to the `[]` operator:

```
p a[0, 1].eval() # => [1, 3]
p a[1, 1].eval() # => [2, 4]
```

The index doesn't have to be constant, instead an index variable can be supplied to the `[]` operator, and then specified when `eval` is called

```
i = Index::new # initialize an index variable
a_row = a[i, 0]
a_col = a[i, 1]

p a_row.eval(i=>0) # => [1, 2]
p a_row.eval(i=>1) # => [3, 4]

p a_col.eval(i=>0) # => [1, 3]
p a_col.eval(i=>1) # => [2, 4]
```

The final example shows how indexing operators can be chained:

```
# indexing can be done several times
# as long as the MDO still has dimensions

p a[0][0].eval() # => 1
p a[0][1].eval() # => 2
# etc...
```

Kronecker delta (`kronecker`)

The Kronecker delta can be produced with the constructor `MDO::kronecker`. It does not provide much functionality when used manually, but it can be used to construct identities of arbitrary rank. The zero, first and second order identities can be defined using one Kronecker delta broadcasted to their respective dimensions.

```
a = MDO::kronecker

p a.eval() # => 1.0
p a.bc(2).eval() # => [1.0, 1.0]
p a.bc(2,2).eval() # => [[1.0, 0.0], [0.0, 1.0]]
```

In Section 2.4 it was demonstrated that

$$\frac{\partial a_{i_1 i_2 \ldots i_r}}{\partial a_{j_1 j_2 \ldots j_r}} = \delta_{i_1 j_1} \delta_{i_2 j_2} \ldots \delta_{i_r j_r} \tag{3.28}$$

Equation (3.28) is equivalent to an identity[13] of rank r. It can be constructed by multiplying Kronecker delta functions. Using the rank three identity as example, $\delta_{ijk} = \delta_{ij}\delta_{ik}$ for can be written as:

```
i3 = (MDO::kronecker(2) * MDO::kronecker )
```

Writing `i3.bc(3,3,3).eval` outputs

```
[[[1.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
[[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 0.0]],
[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 1.0]]]
```

The above is a rank 3 identity with `1.0` along the main diagonal

`MDO::kronecker` without any arguments compares the first and last index; i and k. `MDO::kronecker(2)` compares the first and the *second* index (i and j) because of the argument: `2`. Creating the fourth order identity δ_{ijkl} is a matter of defining $\delta_{ij}\delta_{ik}\delta_{il}$

```
δij = MDO::kronecker(2) # compare the first and second index
δik = MDO::kronecker(3) # compare the first and third index
δil = MDO::kronecker # compare the first and last index
```

Giving:

```
i4 = (MDO::kronecker(2) * MDO::kronecker(3) * MDO::kronecker )
```

3.6 Taking the gradient of an MDO object

So far, scalar derivatives have been considered (see Section 3.3). As implied by the title of this thesis, finding an analytic expression for the *gradient* is the real objective. The gradient is the derivative of a function with respect to either all or a set of the free variables. In the context of MDOs, this can be seen as the derivative of one MDO function with respect to a first order MDO containing the set of free variables one wants the derivative with respect to. The title of this thesis also says "Gradient Calculations to *Arbitrary Order*". This means that taking the gradient of an `MDO` object better result in a new `MDO` object, so that the gradient may be taken consecutively.

Taking the gradient of a function of multiple variables results in the rank increasing by one, it is therefore natural that taking the gradient of an MDO function results in the root node becoming a node with `operator.class == MDOop` (the operator for adding a dimension). Each element of this MDO contains the scalar derivative with respect to the free variable occupying that position in the free variable MDO. Written out as a code example, this means that, if `f` was previously defined as a function of `x, y` and z, then

[13] If we take the rank r identity to mean a rank r MDO with elements $a_{i_1 i_2 \ldots i_r}$ such that $a_{i_1 i_2 \ldots i_r} = 1$ if $i_1 = i_2 = \ldots = i_r$ and $a_{i_1 i_2 \ldots i_r} = 0$ otherwise.

```
v = MDO::MDO(x, y, z)

dfdv = f.grad(v)
```

is equivalent to

```
dfdv = MDO::MDO( f.grad(x), f.grad(y), f.grad(z) )
```

That is how the gradient operator works when supplied a rank 1 MDO as argument. The resulting function `dfdv` is also of class `MDO` and the `grad` function can be called subsequently if a higher order derivative is desired.

3.7 Indefinite size MDOs

In thermodynamic modelling it is often the case that when applying a model, one wants not only to specify values of parameters and free variables, but also the actual number of free variables. Multicomponent systems can in fact have any number of components. Changing the number of free variables affects the dimension sizes of gradients and other MDOs associated with the system. It is undesirable to code up the model each time the number of components change. For that reason, a way of specifying variable dimension sizes at run-time is necessary.

The simplest way of solving this problem is to allow for dimension sizes as free variables. As other free variables, these would have to be specified when the MDO is evaluated. The class of variable used for dimension sizes is called `Dim` for distinction purposes, but it behaves like the `MDO` class. It should be mentioned that the unspecified dimensions are never *infinite*, but *indefinite*, and could in theory be of arbitrary length. In the following example, the dimension length argument to the `bc` (broadcasting) function is replaced by a dimension variable reference.

```
c = Dim::new
n = MDO::const(1.0).bc(c)

p n.eval(c=>2) # => [1.0, 1.0]
p n.eval(c=>3) # => [1.0, 1.0, 1.0]
p n.eval(c=>4) # => [1.0, 1.0, 1.0, 1.0]
```

When specifying a new free variable, its dimension can be given as an argument to the initializer `MDO::new`. The following example implements

$$\begin{aligned} n &= [n_1, n_2, ..., n_c] \\ f &= 2n \end{aligned} \tag{3.29}$$

and evaluates $f(n = [1, 2])$ then $f(n = [1, 2, 3])$. Specifying the value of a rank > 0 MDO's in the `eval` function is done through a nested list:

```
c = Dim::new
n = MDO::new(c)

f = 2.0*n

p f.eval({n=>[1.0, 2.0]}) # =>[2.0, 4.0]
p f.eval({n=>[1.0, 2.0, 3.0]}) # =>[2.0, 4.0, 6.0]
```

The value of `c`, 3 in the latter evaluation call, could have been specified in the hash table {`n=>[1.0, 2.0, 3.0], c=>3`}, though it would be redundant. Since the length of the array `[1.0, 2.0, 3.0]` is three, `c` is automatically given the right value. Rank > 1 free variables are also supported to a limited degree (if the dimension lengths are all the same).

```
c = Dim::new
n = MDO::new(c, c)

f = 2.0*n

p f.eval(n=>[[1.0, 2.0], [3.0, 4.0]])
# =>[[2.0, 4.0], [6.0, 8.0]]
```

It is worth mentioning that differentiation with respect to free variables of unspecified lengths is not as straight-forward as differentiation with respect to constant length MDOs. When a new free variable of unspecified length is created

```
c = Dim::new
n = MDO::new(c)
```

the variable `n` contains a reference to a node with operator class `MDOop` which has a child node being a scalar independent variable to represent an element of `n`. Say the function, `f = n.sum` is differentiated. Its graph representation is shown in Figure 3.8. Taking the derivative would involve writing:

```
dfdn = f.grad(n)
```

The gradient call adds an `MDO` node with operator class `MDOop` as the root node. Then it passes right through the summation operator (not affected by differentiation), and when finally `grad` is called on the elements of `n` itself (the bottom node), it replaces it with a Kronecker delta, rather than `1.0` as it would if it was a scalar.

In Section 2.4, the Kronecker delta was introduced to replace dn_i/dn_j. The Kronecker delta compares two indices: The index attempting to access element n_i and the index of the differentiation variable n_j. The `kronecker` node needs a way in which to identify these two indices. Figure 3.9 shows the layout of `dfdn`.

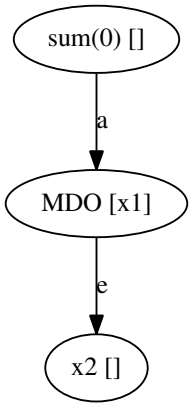

Figure 3.8: A graph representation of the function $f(\mathbf{n}) = \sum_{i=0}^{c} n_i$, where $\mathbf{n}$ is a vector with c elements. The dimensions are shown in square brackets to the right of the node type. The bottom node labelled "x2" represents an element in the free variable $\mathbf{n}$. The variable $\mathbf{n}$ is represented by the node labelled "MDO". "x1" is the name generated for the dimension length variable c. The zero argument to the sum indicates that the first (and only) dimension should be summed over.

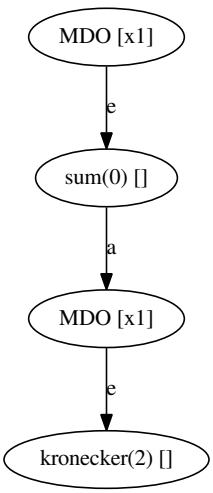

Figure 3.9: A graphical representation of the derivative of the function $f(\mathbf{n}) = \sum_{i=0}^{c} n_i$ with respect to $\mathbf{n}$, where $\mathbf{n}$ is a vector with c elements. The derivative shown by the graph has elements $\partial f/\partial n_j = \sum_{i=0}^{c} \delta_{ij}$ which is equivalent to $\partial f/\partial n_j = 1$. The dimensions are shown in square brackets to the right of the node type. The bottom node labelled "kronecker" represents Kronecker's delta. "x1" is the name generated for the dimension variable c. The zero argument to the sum indicates that the first (and only) dimension should be summed over.

The node labelled "kronecker" has an argument, 2. This indicates the position of the index j corresponding to the differentiation variable and was previously discussed in Section 3.5.

3.8 Hybrid MDOs

The hybrid is another variation of the indefinite size MDO, where the first part of the dimension contains references to specific elements, for example: In thermodynamics the list of free variables often take the form of $\mathbf{x} = [T, V, n_1, n_2, ..., n_C]$ or $\mathbf{x} = [T, p, n_1, n_2, ..., n_C]$. The first two elements, T and V or T and p, have less in common than the rest of the elements n_i, yet they are all free variables. So taking the gradient of Helmholtz energy or Gibbs energy would mean to differentiate with respect to $\mathbf{x}$. The length of $\mathbf{x}$ is $2 + C$ i.e. it depends on the length of $\mathbf{n}$. This is where dimension sizes behaving like MDOs become useful; instead of using a free variable for the dimension size, an expression is used. Here the dimension length of $\mathbf{x}$ is not a free variable, but a function of C.

Setting up `x` as a hybrid length `MDO` is simple:

```
c = Dim::new
n = MDO::new(c) # create a free variable MDO with size c

t = MDO::new
v = MDO::new

x = MDO::MDO(t, v) << n

point = {t=>298.0, v=>0.01, n=>[1.0, 2.0, 3.0]}
p x.eval(point) # => [298.0, 0.01, 1.0, 2.0, 3.0]
```

The Ruby "append" operator `<<` is used for this function, as it closely matches what is being done: Appending an unspecified dimension length MDO to a constant dimension length MDO. The indefinite part of an MDO can only be added at the end, the reason for this choice is that this layout is the most frequently used in thermodynamics. Other options to consider include the use of the splat operator: `MDO::MDO(t, v, *n)`. This was discarded because the splat symbol, `*`, is not *directly* overloadable in Ruby, and therefore would require an ugly workaround.

Little new functionality was needed to handle the hybrid `MDO`; before proceeding with either `eval` or `diff`, the constant size part would simply be handled as constant size `MDO` while the indefinite size part would be handled as a pure indefinite `MDO`. The graph layout of a hybrid MDO is like a constant dimension MDO, but with the indefinite part as the last element. The graph layout of `x = MDO::MDO(t, v) << n` is shown in Figure 3.10. The two first element nodes represent T and V whilst the last node contains the sub-array $n_1, n_2, ..., n_C$. The dimension length denoted by `x3` is the one defined as `c` in the above example code, while the dimension length `x9` refers to the expression $2 + C$.

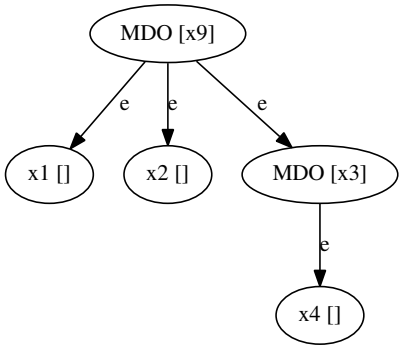

Figure 3.10: The `MDO` graph representation of the hybrid vector $[T, V, n_1, n_2, ..., n_C]$. The nodes labelled "x1" and "x2" refer to T and V while the sub-graph "MDO [x3]" and down contains the sub-array $n_1, n_2, ..., n_C$. "x3" is the sub-array length C, while "x9" refers to the length of the total and is not a free variable, but an expression: $2 + C$. "x4" represents and element n_i.

Reduction

The `grad` method applies differentiation rules without any simplification of the resulting expression. Taking successive gradients of the same function therefore produces large graphs which take a long time to evaluate. The solution to this problem is to do simplifications for each call to the `grad` function. In particular, differentiating produces a lot of ones and zeros. Ones, where the free variable is differentiated with respect to itself, and zeros where other variable are differentiated with respect to the free variable.

As an example, take

```
f=x**2+b
```

Differentiating with respect to `x` and using only the raw differentiation rules described in Section 3.4, gives the result:

```
dfdx = (((2.0*x)*1.0)+0.0)
```

The constant `1.0` arises because the chain rule is used to differentiate `x**2.0` where `x` itself is the inner function. It is clear that the `1.0` can be omitted alongside with the `0.0`, but the computer needs specific rules for this simplification. The `clean!`[14] method is automatically called when the gradient is taken with `grad`. All the rules accounted for have been summarized in Table 3.1

[14]The "bang" symbol ! is used in Ruby to indicate that a method changes the object itself instead of creating a new one and returning a reference to the new object. When `clean!` is called, it simplifies the `MDO` object from which it was called rather than creating a simplified `MDO` object and returning that.

Table 3.1: Rules for simplifying MDO expressions

#	Description	Example
1.	Plus zero can be omitted	$a + 0 = a$
2.	Minus zero can be omitted	$a - 0 = a$
3.	Any lambda operator with only constant children can be evaluated	$5 + 2 = 7$
4.	Anything multiplied by zero is zero	$a \times 0 = 0$
5.	Anything multiplied by one is itself	$a \times 1 = a$
6.	Zero divided by anything is zero	$0/a = 0$
7.	Anything divided by one is itself	$a/1 = a$
8.	The logarithm of one is zero	$\ln 1 = 0$
9.	The exponential of zero is one	$\exp(0) = 1$
10.	Anything to the power of one is itself	$a^1 = a$
11.	The sum of zero is zero	$\sum_{i=0}^{c} 0 = 0$

External software, like Maple, could have been used for simplification. This has not been implemented, but the opportunity is discussed in Chapter 5

Chapter 4

Results

In this chapter the `MDO` class is used to implement a thermodynamic model for a hydrocarbon mixture using the RK-EOS model described in Section 2.5. Section 4.1 introduces the model system. The reader will be guided through an algebraic implementation of the the model using `MDO` objects: In Section 4.2, each line of code is explained and mathematical analogies are shown to provide the reader with enough context to follow the reasoning.

In order to confirm that the project goals were met, Section 4.3 attempts to verify that gradients of arbitrary order (at least up to fourth) are calculated correctly. To give an example of a thermodynamic application, the model is used to produce a phase diagram in Section 4.4. All derivatives needed for Newton iteration were found automatically as `MDO` objects.

4.1 The model system

The model in Section 2.5 was applied to a hydrocarbon mixture of methane, ethane and propane. The system specific parameters are listed in Table 4.1.

Table 4.1: System specific parameters. Critical temperature and pressure found from Lange and Dean (1979). The heat capacity parameters are found from Berkeley Gas Research Institute (1999). All binary interaction parameters were assumed to be zero.

Property		Value			Units
		Methane	Ethane	Propane	
Mole num.	n_i	80.0	60.0	40.0	mol
Crit. temp.	T_c	190.6	305.3	369.6	K
Crit. pres.	P_c	4.61	4.91	4.25	MPa
Heat	A	42.82	35.68	7.76	J/(molK)
capacity	B	-0.11	-4.57×10^{-2}	0.22	J/(molK^2)
parameters	C	4.09×10^{-4}	-4.98×10^{-4}	5.08×10^{-5}	J/(molK^3)
	D	-4.03×10^{-7}	-5.89×10^{-7}	-1.83×10^{-7}	J/(molK^4)
	E	1.39×10^{-10}	2.23×10^{-10}	7.91×10^{-11}	J/(molK^5)
Interaction	Methane	0.0	0.0	0.0	-
parameters	Ethane	0.0	0.0	0.0	-
k_{ij}	Propane	0.0	0.0	0.0	-

As mentioned in Section 2.5, the expression for the heat capacity of component i is:

$$c^0_{p,i}(T) = A_i + B_iT + C_iT^2 + D_iT^3 + E_iT^4 \tag{4.1}$$

All interaction parameters were set to zero. The reason that they are included at all is to show the `MDO` class' ability to handle rank > 1 free variables.

4.2 Implementation using MDO objects

In this section the thermodynamic model described in Chapter 2 is implemented using the `MDO` class. The complete implementation can be found in Appendix B. Some code comments have been omitted to enhance readability. The code lines are presented in order, with descriptions in between explaining what the code does.

The first line

```
require './mdo_main.rb'
```

is the command to import the file `'mdo_main.rb'` which contains the source code for the `MDO` class. The next part is:

```
r = MDO::const(8.314)
c = Dim::new # the number of components defined as a dimension

t = MDO::new # temperature
v = MDO::new # volume
```

First the universal gas constant R, here given the identifier `r`, is initialized and given a value. This `MDO` object now represents a *constant* node. The number of components, C, is initialized as `c`; a free dimension-variable. Similarly, the temperature and volume variables are named `t` and `v` respectively and initialized as *independent* variable `MDO` objects by use of the `MDO::new` constructor. Since no arguments are given to `MDO::new`, `t` and `v` are rank 0 MDO's (scalars).

The next few lines look a little bit different.

```
n_vec = MDO::new(c) # mole number vector

tc_vec = MDO::new(c) # critical temperature vector
pc_vec = MDO::new(c) # critical pressure vector
```

The free variable $\mathbf{n}$ is initialized with size C, which gives it rank 1. So far, the free variables are $T, V, \mathbf{n}$. No surprises there, seeing as they are the canonical free variables of Helmholtz energy. The subsequent two lines, however, initialize the critical temperature and pressure, `tc_vec` and `pc_vec`, as free variables with length C. Since we want the model to be able to handle different systems at run-time, system specific parameters like T_c and P_c must be initialized as free variables. Since each component has a critical temperature and pressure, `tc_vec` and `pc_vec` are given size C.

The next few lines implement the definition of the a and b parameters for the RK-EOS

```
b_vec = 0.0867*r*tc_vec/pc_vec
a_vec = 0.4278*r**2.0*tc_vec**2.5/pc_vec
```

These two `MDO` expressions can be recognised as Equations (2.45) and (2.46)

$$b_i = 0.0867RT_{c,i}/p_{c,i}$$

$$a_i = 0.4278R^2T_{c,i}^{2.5}/p_{c,i}$$

It is important to reinforce that nothing is actually being calculated yet. The only thing happening is that `MDO` object relations are set up, ready for evaluation or differentiation. Working with vectors and scalar operators as in the above lines of code, results in element by element operations. Writing `b_vec = 0.0867*r*tc_vec/pc_vec` is equivalent to $b_i = 0.087RT_{c,i}/p_{c,i}$. Here, the rank 0 MDO, `r`, is multiplied with the rank 1 MDO `tc_vec`. This results in `r` to be default broadcasted into the dimensions of `tc_vec`: Every element of `tc_vec` will be multiplied by the value of `r` upon evaluation.

The next bit of code creates a new independent variable representing k_{ij}. Since each interaction parameter is between two components, two indices are needed to specify a value k_{ij}. The interaction parameter matrix is therefore given two dimensions of the same size as $\mathbf{n}$.

```
k_mat = MDO::new(c, c) # interaction parameter matrix
```

This parameter is used in the mixing rule $a_{ij} = (a_i a_j)^{1/2}(1 - k_{ij})$, from Equation (2.44). Implementing that equation is not as straight forward as the previous equations. An element a_i multiplied by an element a_j is not an element-by-element multiplication seeing as the index i may be different from the index j.

The way of solving this is through broadcasting: Call

$$a^*_{ij} = a_i \tag{4.2}$$

and

$$\hat{a}_{ij} = a_j \tag{4.3}$$

two different broadcasted versions of $\mathbf{a} = [a_1, a_2, ..., a_C]$. Following this definition:

$$(a_i a_j) = (a^*_{ij} \hat{a}_{ij}) \tag{4.4}$$

The first broadcast, $a^*_{ij} = a_i$, keeps the first index i and is therefore said to be broadcasted in the second dimension. Using the `MDO` class, this can be specified as `bc(0, c)` where `c` is the length of the added dimension. The second broadcast, $\hat{a}_{ij} = a_j$, only uses the second index j and is therefore broadcasted in the first dimension. Similarly, this can be specified as `bc(c, 0)`. To summarise:

$a_i \rightarrow$`a_vec.bc(0, c)`
$a_j \rightarrow$`a_vec.bc(c, 0)`

Using these two broadcasts, the following `MDO` expression represents $a_{ij} = (a_i a_j)^{1/2}(1 - k_{ij})$:

```
a_mat = (a_vec.bc(0, c)*a_vec.bc(c, 0))**0.5*(1.0-k_mat)
```

In the next two lines, component-specific parameters a_{ij} and b_i are used to set up expressions for the total mixture parameters a and b which appear in the RK-EOS. This is a direct implementation of Equations (2.42) and (2.43):

$$b = \sum_{i=1}^{c} n_i b_i$$

$$a = \sum_{i=1}^{c} \sum_{j=1}^{c} n_i n_j a_{ij}$$

Implemented as `MDO` expressions:

```
b = (n_vec*b_vec).sum
a = (n_vec.bc(0, c)*n_vec.bc(c, 0)*a_mat).sum.sum
```

As can be seen in the expression for `a`; `n_vec` is broadcasted in just the same way as `a_vec` was. The reason being to distinguish n_i from n_j. Given no argument, the `sum` method defaults to summing the outer dimension[1]. Seeing as i and j are interchangeable in the expression for a_{ij} (a_{ij} is an element of a symmetric matrix), it doesn't matter what dimension is summed first.

The next lines of code do not introduce any new concepts. First, the vector $\mathbf{n}$ is summed to create the total mole number $N = \sum_{i=0}^{C} n_i$. Secondly, Equation (2.41) for pressure[2] and Equation (2.61) for the residual Helmholtz energy (here named `f_res`) are implemented.

```
n_tot = n_vec.sum

p_rk = n_tot*r*t/(v-b) - a/(t**0.5*v*(v+b))

f_res = n_tot*r*t*(v/(v-b)).ln+a/(b*t**0.5)*(v/(v+b)).ln
```

The two bottom equations contain only lambda operators and rank 0 (scalar) MDOs. Because of operator overloading, the Ruby syntax itself is used to construct expression graphs.

Now that the *non-ideal* part of the Helmholtz energy function is constructed, it is time for the *ideal* part. First, a great deal of component-specific parameters must be initialized as free variables. As before, the reasoning for making them free variables is that we want the choice of components to remain open until runtime: The functions themselves can be constructed without prior knowledge of the system.

```
a_cp = MDO::new(c)
b_cp = MDO::new(c)
c_cp = MDO::new(c)
d_cp = MDO::new(c)
e_cp = MDO::new(c)
h0 = MDO::new(c)
s0 = MDO::new(c)
```

The first variables, `a_cp` to `e_cp`, are component specific coefficients for the expression for molar heat capacity, $c_{p,i}(T) = A_i + B_iT + C_iT^2 + D_iT^3 + E_iT^4$. Since each components has its set of parameters A-E, the parameters are initialized as 1d MDOs with size corresponding to the number of components C. Similarly, the component standard enthalpy and entropy, called `h0` and `s0` respectively, are initialized the same way as the parameters for the heat capacity polynomial.

Next, the reference pressure and temperature are initialized as constants.

```
t0 = MDO::const(298.0) # K
p0 = MDO::const(1.0e5) # 1 bar in Pa
```

[1]The outer dimension is the one using the first index, for example, the rows in a matrix

[2]The expression for pressure was not actually used in calculation, the derivative of Helmholtz energy with respect to volume was used instead.

This is done in a similar way as R was initialized. Then the integrals $I_{1,i} = \int_{T_0}^{T} c_{p,i}(T)dT$ and $I_{2,i} = \int_{T_0}^{T} c_{p,i}(T)/TdT$ are constructed as:

```
# integral of cp(T) dT
int1 = a_cp*(t-t0) +
      b_cp*(t**2.0-t0**2.0)/2.0 +
      c_cp*(t**3.0-t0**3.0)/3.0 +
      d_cp*(t**4.0-t0**4.0)/4.0 +
      e_cp*(t**5.0-t0**5.0)/5.0

# integral of cp(T)/T dT
int2 = a_cp*(t/t0).ln +
      b_cp*(t-t0) +
      c_cp*(t**2.0-t0**2.0)/2.0 +
      d_cp*(t**3.0-t0**3.0)/3.0 +
      e_cp*(t**4.0-t0**4.0)/4.0
```

Since the `MDO` class only includes support for differentiation and not integration, the integrals must be done by hand first, and then implemented as MDO expressions. The above code resembles the integrals:

$$\begin{aligned}\int_{T_0}^{T} A_i+B_iT+C_iT^2+D_iT^3+E_iT^4dT = \\ A_i(T-T_0)+\frac{B_i}{2}(T^2-T_0^2)+\frac{C_i}{3}(T^3-T_0^3) \\ +\frac{D_i}{4}(T^4-T_0^4)+\frac{E_i}{5}(T^5-T_0^5)\end{aligned} \tag{4.5}$$

and

$$\begin{aligned}\int_{T_0}^{T} \frac{1}{T}(A_i+B_iT+C_iT^2+D_iT^3+E_iT^4)dT = \\ A_i\ln\left(\frac{T}{T_0}\right)+B_i(T-T_0)+\frac{C_i}{2}(T^2-T_0^2) \\ +\frac{D_i}{3}(T^3-T_0^3)+\frac{E_i}{4}(T^4-T_0^4)\end{aligned} \tag{4.6}$$

The goal of constructing these integrals is to express the ideal gas chemical potential. Equation (2.55) for $\mu_i^0(T)$, is finally constructed in parts, and named `mu_vec`.

```
h = h0 + int1
s = s0 + int2
mu0_vec = h - t*s
mu_vec = mu0_vec + r*t*(n_vec*r*t/(v*p0)).ln
```

Finally, the expression for the ideal gas Helmholtz energy, $A^{\text{ig}} = -NRT+\sum_{i=1}^{C}\mu_i^{\text{ig}}n_i$, can be constructed:

```
f_ig = -n_tot*r*t + (mu_vec*n_vec).sum
```

`f_ig` can be added to `f_res` to create the function for Helmholtz energy `f`.

```
f = f_ig + f_res
```

The function for Helmholtz energy is not evaluated directly in the phase equilibrium calculation scheme, rather its derivatives are what we're after. Since the temperature will be assumed a known constant in the model application, the differentiation variables are $V, n_1, n_2, ..., n_c$. Defining these in a one-dimensional MDO is done as follows:

```
x = MDO::MDO(v) << n_vec
```

The first part, `MDO::MDO(v)`, adds a dimension to the `v` variable. It being rank 0, that results in a rank 1 MDO containing one element, `v`. Next, the `<<` operator appends the elements in `n_vec` to the newly created 1d MDO, making `x` resemble $V, n_1, n_2, ..., n_c$.

Now we can differentiate with respect to `x` by supplying it as argument to the `grad` function

```
y = f.grad(x)
dydx = y.grad(x)
```

`y` is the gradient of Helmholtz energy

$$\frac{\partial A}{\partial \mathbf{x}} = \left[\frac{\partial A}{\partial V}, \frac{\partial A}{\partial n_1}, \frac{\partial A}{\partial n_2}, ..., \frac{\partial A}{\partial n_c}\right] \tag{4.7}$$

Also known as $[-p, \mu_1, \mu_2, ..., \mu_c]$. The second derivative `dydx` is needed in the Newton iteration scheme. Both `y` and `dydx` are now analytic functions, and can be evaluated for a system with any number of components at any temperature, volume and composition.

Now that all necessary functions have been created, what remains before they can be evaluated is to specify the parameter values. This is the break between where the model is declared and what is called "run-time".

```
# CH4 CH3CH3 CH3CH2CH3
tc_vals = [ 190.6 , 305.3 , 369.552 ] # Kelvin
pc_vals = [ 46.1e5 , 49.0e5 , 42.4924e5 ] # Pascal

k_vals = [ [0.0, 0.0, 0.0], # interaction parameter values
           [0.0, 0.0, 0.0], # set to zero
           [0.0, 0.0, 0.0] ]

# cp parameters from 300-1000K
# Cp = A + B*t + C*t^2 + D*t^3 + E*t^4
a_cp_vals = [ 42.8161, 35.6789 , 7.76157 ]
b_cp_vals = [ -0.113661 , -4.573982e-2 , 0.219694 ]
c_cp_vals = [ 4.08883e-4, 4.983730e-4 , 5.07651e-5 ]
d_cp_vals = [ -4.0301535e-7 , -5.890189e-7 , -1.827209e-7 ]
e_cp_vals = [ 1.38589e-10 , 2.2338535e-10, 7.91070889e-11 ]

h0_vals = [ -78870.0 , -84000 , -104700 ] # J/mol 298K (g) 1
bar
s0_vals = [ 186.25, 229.6 , 269.91 ] # J/(K mol) 1 bar 298K (g)
```

The parameter values are given on nested list form. It is important that their rank matches exactly that of the free variables which the values are specified for. For example, component specific parameters are all a single list with three numbers, while the interaction parameter values `k_vals` are specified in a list within a list, representing a rank 2 MDO. The dimension lengths of `k_vals` also matches the number of components for this system (three).

The dimension length `c` is not specified explicitly, as its value, 3, is deduced from the length of the lists by the `eval` function. Since the argument to the `eval` function is a hash table associating the variables with their values, this has to be created next.

```
params = {tc_vec=>tc_vals, pc_vec=>pc_vals,
          a_cp => a_cp_vals,
          b_cp => b_cp_vals,
          c_cp => c_cp_vals,
          d_cp => d_cp_vals,
          e_cp => e_cp_vals,
          h0=>h0_vals, s0=>s0_vals,
          k_mat=>k_vals}
```

The most important variable values however, T, V, and $\mathbf{n}$ have not been specified yet. They are the canonical free variables of the Helmholtz energy function, and will vary for different applications. T will however be assumed known and constant , while $\mathbf{x} = [V, n_1, n_2, ..., n_c]$ will be iterated upon. To give an example, `y` and `dydx` can now be evaluated for some example choices of T, V, and $\mathbf{n}$. Since the parameters have been specified using three components, $\mathbf{n}$ must correspondingly have three values.

```
t_val = 280.0 # K
v_val = 0.07 # m3
n_vals = [ 80.0 , 60.0 , 40.0 ] # moles

p y.eval(params.merge({t=>t_val, v=>v_val, n_vec=>n_vals}))[3]
p dydx.eval(params.merge(t=>t_val, v=>v_val, n_vec=>n_vals))
```

The incrementation step of the iteration scheme (Equation 2.66) reads:

$$\Delta \mathbf{x}_k^v = \left[\left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^v \bigg|_{\mathbf{x}_k^v} + \left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^l \bigg|_{\mathbf{x}_k^l} \right]^{-1} \left(\mathbf{y}^l(\mathbf{x}_k^l) - \mathbf{y}^v(\mathbf{x}_k^v) \right)$$

If V^v is represented by a numeric value `v_val_v` and $\mathbf{n}^v$ by a list of numbers `n_vals_v`, then evaluating

$$\left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^v \bigg|_{\mathbf{x}_k^v}$$

is a matter of writing

```
dydx.eval(params.merge({t=>t_val, v=>v_val_v, n_vec=>n_vals_v})
```

Similary, if V^l was called `v_val_l` and $\mathbf{n}^v$ was called `n_vals_v`, then

$$\left(\frac{d\mathbf{y}}{d\mathbf{x}^\top} \right)^l \bigg|_{\mathbf{x}_k^l}$$

can be evaluated by writing

```
dydx.eval(params.merge({t=>t_val, v=>v_val_l, n_vec=>n_vals_l})
```

The only difference between the two above cases is that the values of the free variables $V, \mathbf{n}$ are given as the vapour values and liquid value respectively.

Being able to evaluate $\mathbf{y}(\mathbf{x})$ and its derivative for both phases, the iteration scheme can proceed to solve for $\mathbf{n}^l$, $\mathbf{n}^v$, V^l, V^v. The source code for the iteration scheme can be found Appendix C.

4.3 Verifying the gradient calculation

In order to verify that the gradient expressions found by the `grad` function are correct, they were compared with numerical estimates. Recursive central difference was used to estimate arbitrary order derivatives. To estimate the next level derivative, the following formula was used:

$$\frac{\partial f}{\partial x} = \frac{f(x + \Delta x/2) - f(x - \Delta x/2)}{\Delta x} \tag{4.8}$$

[3] `params.merge(...)` means that the hash table specifying parameter values is merged with the one specifying free variables `{t=>t_val, v=>v_val, n_vec=>n_vals}`. Together they fully specify the state of the evaluation.

Where f could be the undifferentiated function itself or a lower derivative. The accuracy of the method was not of great importance, as the intention is to compare with analytically found gradients. If the numbers lie in the same neighbourhood, that will be enough reason not to doubt the analytic derivatives.

The test case used the function for Helmholtz energy derived in Section 2.5. The first, second, third and fourth order gradients (∇A, $\nabla\nabla A$, $\nabla\nabla\nabla A$ and $\nabla\nabla\nabla\nabla A$) were compared at the point $T = 280$ K, $V = 0.07$ m^3, $\mathbf{n} = [80.0, 60.0, 40.0]$ mol, using methane, ethane and propane as components in that order. The gradient operator was defined as

$$\nabla = \left[\frac{\partial}{\partial V}, \frac{\partial}{\partial n_1}, \frac{\partial}{\partial n_2}, \frac{\partial}{\partial n_3}\right] \tag{4.9}$$

Note that constant temperature was assumed.

The calculated gradients were compared element by element, for example, the fourth order gradient contained $4 \times 4 \times 4 \times 4 = 256$ elements. Each element of the analytic calculation was compared with the numerical. The comparisons were successful for all gradient orders. The analytic gradient elements never deviating more than 1.1%[4] from the numerical gradient. This deviation can be explained by truncation error in the numerical estimates. Therefore it can be safely assumed that the analytic gradients are calculated correctly.

Appendix D shows the relevant Ruby script files and detailed results of the comparison. The performance of evaluating the analytically derived gradients is, on the other hand, very bad compared to the numerical estimates. For example: The 3rd order numerical gradient takes ~ 0.5 seconds to evaluate, whereas the 3rd order analytic gradient takes ~ 20 seconds.

4.4 Tracing the two-phase boundary

In this section, a T, V-diagram is produced showing a part of the two-phase region for the system described in Section 4.1. The two-phase region can be found as the set of values for T and V (total volume) for which the equilibrium calculation converges. However, the phase-equilibrium calculation does not always converge even inside the two phase region: The Newton iteration has to be initialized with a guess which is close to the solution.

To assure convergence in the entire two-phase region, phase equilibrium calculation was started out at a T, V-point known to be well within the two phase region. When this equilibrium was found, it was used as a guess to calculate the equilibrium at a nearby point. The equilibrium found at this point was used as guess to calculate a new point, and so on. If a point did not converge, a point closer to the guess was attempted. Scripts to produce the diagram are supplied in Appendix E. The resulting phase envelope can be seen in Figure 4.1

[4] This deviation was seen in the fourth order derivative, and is not of note given the uncertainty in the numerical algorithm.

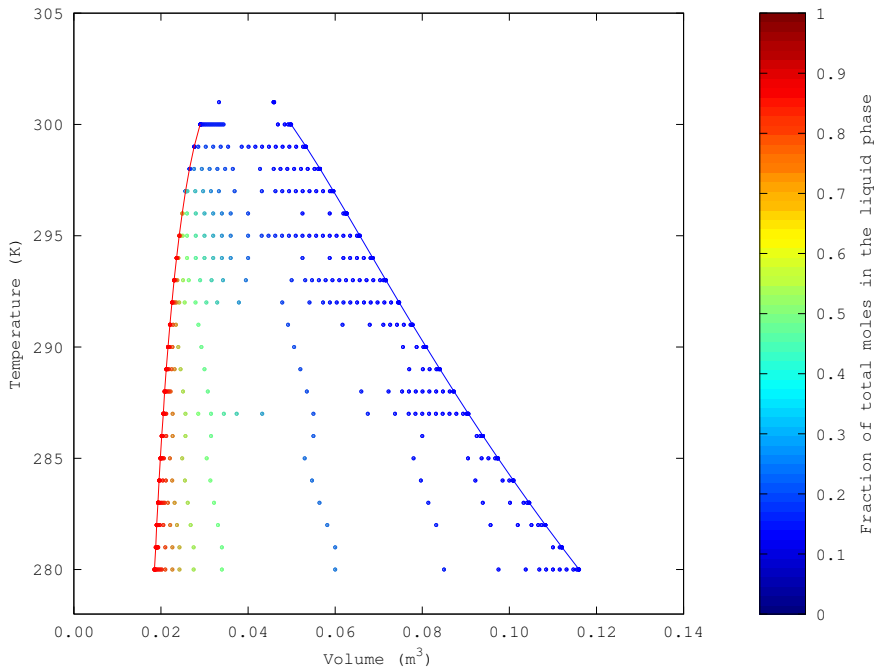

Figure 4.1: Phase diagram for a system of 80 moles of methane, 60 moles of ethane and 40 moles of propane. The volume and composition in gas and liquid phase was found at each point. The colors indicate volume fraction of liquid phase. Red means mostly liquid, blue means mostly gas. Since this mixture was high in the lighter components, the liquid phase is not present in high volume other than close to the pure-liquid region (to the left of the two-phase region). The top of the phase envelope (close to the critical point) could not be calculated due to convergence problems.

The points shown in the figure are the points where phase equilibrium calculation was successfully attempted. Close to the critical point, at around 302 K and 0.045 m^3 the iteration scheme has trouble differentiating between the two phases, as the mixture is about to become one homogeneous supercritical fluid.

The data for Figure 4.1 is produced entirely through Ruby scripts, but has been visualized using Octave. Producing the data takes approximately two hours on a mid-range CPU. No rigorous profiling has been done, but experience indicates that much time is spent evaluating the `MDO` object `dydx` (the Jacobian in the Newton-iteration). The reduction scheme outlined in Section 3.8 helps, but could benefit from interfacing with another computer algebra system, like Maple, or Mathematica. This, and being able to export the functions `y` and `dydx` to a faster language, like C, would go a long way towards reducing time spent on evaluation. Options for increasing performance are discussed further in Chapter 5.

Convergence

In Section 2.5 it was mentioned that Newton's method converged quadratically. To confirm this, the residual norms (L2-norm $\|\mathbf{g}\| = \sqrt{\sum g_i^2}$) in iteration step number $k+1$ have been plotted against the residual norms in the previous step k. Figure 4.2 shows the development of the residual norm for a sample phase calculation: The slope of the curve is 2, confirming that the method is indeed quadratic. The very first iteration (the flat part to the right in Figure 4.2) did not yield as large an increase in precision as the subsequent iteration steps. Sometimes, a bad initial guess leads the free (extensive) variables to drop below zero in the first step (this is unphysical). Step-size control reduces $\Delta\mathbf{x}$ to avoid this but at the same time slows convergence.

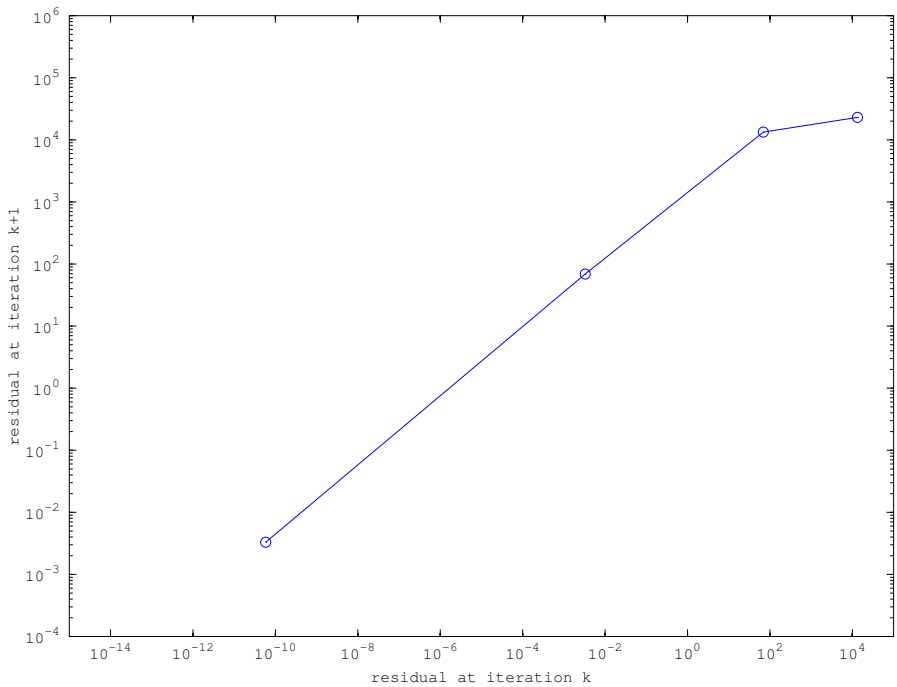

Figure 4.2: Shows the residual in the next step $k+1$ as a function of the residual in the last step k. The axes are logarithmic. The slope of the curve is almost exactly 2, confirming that Newton's method is of quadratic convergence order. The iteration starts with the right-most point

Chapter 5

Discussion

This chapter deals with some of the loose ends regarding the `MDO` class. Did the algebraic approach provide a benefit compared to a numerical approach? Simplification is one issue that could be improved: Applying the chain rule to differentiate a function without any simplification results in long expressions. Next we look at the missed opportunities provided by rank > 1 differentiation variables in the context of sensitivity analysis. Legendre transforms is another part of thermodynamics that is not covered by the `MDO` class. This is discussed as well as the opportunity to export code to other languages.

Benefits of the MDO system

If the alternative is to find partial derivative functions by hand, then being able to find analytic derivatives at the call of a (`grad`) method has its clear advantages. However, when using Newton's method it is common to estimate the Jacobian numerically. New names are given to these types of methods depending on the numerical differentiation scheme: The secant method, for example, is Newton's method using "backwards-difference" to find the Jacobian (Kelley, 2003). Calculating the Jacobian numerically is a slow, but relatively straight-forward process (Arrillaga and Watson, 2004). There will also be numerical error in the Jacobian as opposed to an analytically calculated Jacobian. This leads to slower convergence, the secant method for example has super-linear convergence order, but not quite quadratic (Kelley, 2003). However it should be emphasised that convergence order does not change the accuracy of the solution approximated by Newton's method, it only changes the number of iterations needed to get there.

When comparing the analytic gradient calculation with the finite difference method (Section 4.3), there was a significant gap in performance: The numerical gradient was much quicker to evaluate. For the application of producing a phase diagram, using numerical derivatives would have resulted in a much faster overall algorithm. For speed, using `MDO` objects is clearly not advantageous. However, having implemented the thermodynamic model in Section 4.2 once means that it can be reused for a new system with any number and choice of components.

Reduction

Reduction in computer algebra means to simplify an expression. As mentioned in Section 4.4, calling the `eval` method is a time consuming operation for complex MDOs. A complex MDO expression has very many nodes (operators), some of these might be eliminated by simplification, something which is quite extensive to implement algorithmically. Luckily, software like Maple, Mathematica, Reduce among others are very efficient at simplifying mathematical expressions. Though, they handle for the most part scalar expressions. If the scalar (lambda) parts of an MDO expression could be separated out, then they could be exported as strings[1], simplified, then returned. Figure 5.1 attempts to illustrate the possible flow of a call to the `grad` method using external software simplification:

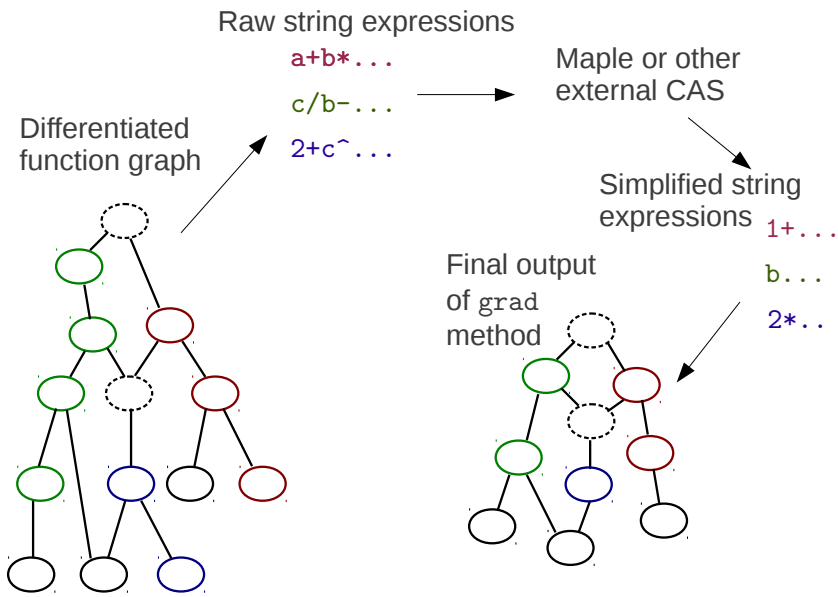

Figure 5.1: The chart illustrates a way of simplifying an MDO graph by using external software. To the left is a complex graph. Structure nodes are shown as dashed circles, while lambda (scalar operator) nodes are colored, one color for each lambda function (they are separated by structure nodes). Most computer algebra systems can only handle scalar expressions: the lambda functions can be exported to string format. They can then be simplified by external software. The simplified string expressions can be read back to Ruby to recreate parts of the MDO graph. The simplified expressions use fewer nodes.

[1]String is an series of characters, for example $\frac{a+b}{c^2}$ could have a string representation of `"(a+b)/c^2"`

The main hurdle to overcome when interfacing with external software is the input-output format. The `MDO` class would have to have a method for converting an expression graph into a set of lambda function strings, then reading back the simplified string expressions. In Figure 5.1, the dashed nodes represent structure operators. Lambda nodes are the one in between. Structure operators separate lambda functions, shown as green red and blue nodes. The black nodes represent free variables. Since external software tend to deal with scalar functions, the lambda parts have to be exported separately, simplified, and reintroduced to the graph, in between the structure operators.

Løvfall (2008) used a notation where the lambda function was strictly separated from the structure of an algebraic object. For the purpose of exporting lambda functions for simplification, this approach is advantageous. Using `MDO` objects there is some extra work involved in separating lambda nodes from structure nodes. There should be little trouble recreating `MDO` expressions from external input, seeing as the operators `+-*/` have been overloaded, making the `MDO` expression syntax similar to general mathematical notation (with some exceptions, post-fix (...).ln instead of ln(...) being one of them).

Differentiation variables above rank one

The basic approach to sensitivity analysis of a problem on the form

$$F(y, x, p) = 0 \tag{5.1}$$

is to find the derivative of the above system with respect to all parameters (Maly and Petzold, 1996). In thermodynamics, the interaction parameters $\mathbf{k} = [k_{ij}]$ are an example of a rank 2 parameter. Doing a sensitivity analysis of the model described in Section 2.5 would involve differentiating with respect to $\mathbf{k}$ something which is not currently possible using `MDO` objects.

In Section 2.4, the Kronecker delta (currently implemented) is shown to be sufficient for describing the derivative of a rank r MDO with respect to itself:

$$\frac{\partial a_{i_1 i_2 \ldots i_r}}{\partial a_{j_1 j_2 \ldots j_r}} = \delta_{i_1 j_1} \delta_{i_2 j_2} \ldots \delta_{i_r j_r} \tag{5.2}$$

Therefore, implementing support for higher rank differentiation variables (than rank 1) does not require implementation of any new operators. The `grad` function, however, would need to be adapted. For instance, while differentiating with respect to a rank 1 MDO adds one dimension to a function, differentiating with respect to a rank 2 MDO adds two new dimensions[2]. Due to time constraints and lack of an immediate application, rank > 1 differentiation variables have not been implemented

[2]The number of dimensions added when differentiating equals the rank of the differentiation variable: $\frac{\partial f_{ijk}}{\partial x_{lm}} = \left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ijklm}$: Here a rank 3 MDO $\mathbf{f}$ is differentiated with respect to a rank 2 MDO $\mathbf{x}$. The derivative has rank 5.

Variable transforms

Producing a phase diagram is but one application using the `MDO` class. Any model can be implemented as long as an *explicit* expression for an energy function, like A or G can be formulated. Because of the gradient-taking capabilities of the `MDO` class, derivative properties, like μ_i, will follow. The major limitation of the `MDO` class is its inability to handle *implicit* expressions. Legendre transforms, for example, requires a change of variables. Rearrangement of `MDO` expressions and implicit differentiation is not possible. In this regard, traditional computer algebra systems hold the upper hand.

Say, for example, that an explicit function for $A(T, V, \mathbf{n})$ has been defined and called `f` using free variables initialized as `t`, `v` and `n_vec`. To derive an expression for U the Legendre transform of A is taken with respect to $-T$:

$$\begin{aligned} U &= A - (-T)\left(\frac{\partial A}{\partial(-T)}\right)_{V,\mathbf{n}} \\ &= A + TS \end{aligned} \tag{5.3}$$

This could be implemented as:

```
s = -f.grad(t)
u = f + t*s
```

The internal energy U could be evaluated, but the function `u` would still use the free variables `t`, `v` and `n_vec`. This is a problem, because we want to be able to produce partial derivatives of U with respect to its canonical variables, e.g. $T = (\partial U/\partial S)_{V,\mathbf{n}}$.

What could solve the above issue is if it was possible to rearrange expressions. Then T could be made a function of S, and substituted in. In terms of a graph representation this is *inversion*: What was previously a leaf node becomes the root. Operators would be inverted as well to emulate changing sides in the equation[3]. Rearrangement, however, is not possible in general[4], therefore we are left with the option of handling implicit functions.

We want to evaluate and differentiate $U(S, V, \mathbf{n})$, but we are given are the functions $A(T, V, \mathbf{n})$ and $S(T, V, \mathbf{n})$. $U(S_0, V_0, \mathbf{n}_0)$ can be evaluated by finding T such that

$$S(T, V_0, \mathbf{n}_0) = S_0 \tag{5.4}$$

then calculate

$$U = A(T, V_0, \mathbf{n}_0) + TS(T, V, \mathbf{n}_0) \tag{5.5}$$

This way S is emulated as free variable. Differentiating U with respect to S could be done via the chain rule

$$\frac{\partial U}{\partial S} = \frac{\partial U}{\partial T}\left(\frac{\partial S}{\partial T}\right)^{-1} \tag{5.6}$$

[3]For example: $a = bc$ rearranged to $c = a/b$, the multiplication operator was inverted to a division operator because b moved to the other side of the equation

[4]For example, you cannot rearrange $x = t + \ln t$ to $t = ...$

Since the code is able to produce U and S as a function of $T, V, \mathbf{n}$ it can also find the derivatives in Equation 5.6.

Even though the above solution would lead to Legendre transformed functions that could be evaluated and differentiated using their canonical variables, the issue still remains that transforming would add immense complexity to the `MDO` expression. Therefore, implementing the above solution should be done in conjunction with performance enhancing measures, such as improved simplification or the ability to export `MDO` expressions as functions in a faster language.

Code export

Building mathematical models using `MDO` objects in Ruby does not demand a lot of computer resources. *Evaluating* the model is what takes time. After expressions have been built, there isn't any need to keep them in the `MDO` object format. They can be exported as calculation routines to a fast language, like C, or FORTRAN. As ordinary programming functions, they can still be evaluated, just no longer differentiated.

This feature has not been implemented due to time constraints, however it would improve the `MDO`s applicability towards more complex thermodynamic modelling.

Chapter 6

Conclusion

The `MDO` class has been implemented in Ruby. It provides a programming interface for creating mathematical expressions using multidimensional algebraic objects (MDOs): A scalar would be represented by a zero-dimensional MDO, a vector by a one-dimensional MDO, and a matrix by two-dimensional MDO. An MDO may have any number of dimensions (rank) $0, 1, 2...\infty$. Algebraic expressions for the gradients of these MDO expressions can be derived automatically by calling the `grad` method on an `MDO` object. The `grad` method returns a new `MDO`, thus subsequent gradients can be derived, enabling differentiation to arbitrary order. MDO expressions can be evaluated as functions by calling the `eval` method, where the values of parameters and free variables are supplied as arguments.

The intended application of the software is thermodynamic modelling of phase equilibria. The `MDO` class has been used to generate a phase diagram of a natural gas system using the Redlich and Kwong (1949) equation of state. Using the same model, the software has been proven successful in evaluating gradients of Helmholtz energy of order up to fourth order. The reason that it has not been tested further is because of performance issues. With limited capability for simplifying expressions, MDOs gain a lot of complexity as they are differentiated.

The project has been successful in providing usable framework for expressing and differentiating thermodynamic energy functions. Being able to handle many-dimensional algebraic objects, though, has come at the cost of more traditional functionality supplied by other computer algebra systems: Reduction using Maple or other external software would remedy the performance issues and efficient legendre transforms require the ability to change free variables. A good way forward would be to interface with other computer algebra systems: The `MDO` class could handle the multi-dimensional side of things, while other aspects are handled by the external CAS.

Bibliography

J. Arrillaga and N.R. Watson. *Power System Harmonics*. Wiley, 2004. ISBN 9780470871218. URL http://books.google.no/books?id=mDGfHsEoiN8C.

Manuel M. T. Chakravarty. *On the Massively Parallel Execution of Declarative Programs*. PhD thesis, Technischen Universität Berlin, 1997.

A.C. Dimian, C.S. Bildea, and A.A. Kiss. *Integrated Design and Simulation of Chemical Processes*. Computer Aided Chemical Engineering. Elsevier Science, 2003. ISBN 9780080534800. URL http://books.google.no/books?id=VTUl2Yph_sMC.

M.J. Dominus. *Higher-Order Perl: Transforming Programs with Programs*. Elsevier Science, 2005. ISBN 9780080478340. URL http://books.google.no/books?id=4_q8JJWNaTsC.

John W. Eaton. Gnu octave documentation, 2011. URL http://www.gnu.org/software/octave/doc/interpreter/Broadcasting.html.

D. Flanagan and Y. Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008. ISBN 9780596554651. URL http://books.google.no/books?id=jcUbTcr5XWwC.

Joachim Gross and Gabriele Sadowski. Application of perturbation theory to a hard-chain reference fluid: an equation of state for square-well chains. *Fluid Phase Equilibria*, 168(2):183 – 199, 2000. ISSN 0378-3812. doi: 10.1016/S0378-3812(00)00302-2. URL http://www.sciencedirect.com/science/article/pii/S0378381200003022.

T. Haug-Warberg. *Den termodynamiske arbeidsboken*. Kolofon Forlag AS, 2006. ISBN 9788230002056.

Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, September 1989. ISSN 0360-0300. doi: 10.1145/72551.72554. URL http://doi.acm.org/10.1145/72551.72554.

Berkeley Gas Research Institute. Gri-mech version 3.0 thermodynamics, July 1999. URL http://www.me.berkeley.edu/gri_mech/version30/files30/thermo30.dat.

K. Jain. *Numerical Methods For Scientific And Engineering Computation*. New Age International (P) Limited, 2003. ISBN 9788122414615. URL http://books.google.no/books?id=5XappvcENCMC.

S.V. Kedar. *Programming Paradigms And Methodology*. Technical Publications, 2008. ISBN 9788184312966. URL http://books.google.no/books?id=gvm9TPE96t4C.

C.T. Kelley. *Solving Nonlinear Equations with Newton's Method*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics, 2003. ISBN 9780898715460. URL http://books.google.no/books?id=-aYWtUuRlwQC.

N.A. Lange and J.A. Dean. *Lange's Handbook of chemistry*. Number v. 12 in Lange's Handbook of Chemistry. McGraw-Hill, 1979. ISBN 9780070161917. URL http://books.google.no/books?id=lnOeAQAAIAAJ.

K.D. Lee. *Programming Languages: An Active Learning Approach*. Springer London, Limited, 2008. ISBN 9780387794228. URL http://books.google.no/books?id=OuW5dC2O99AC.

Richard Liska et al. Computer algebra, algorithms, systems and applications, February 1999. URL inst.eecs.berkeley.edu/ cs282/sp02/readings/liska.pdf.

B.T. Løvfall. *Computer Realization of Thermodynamic Models Using Algebraic Objects*. Doktoravhandlinger ved NTNU. Norwegian University of Science and Technology, Faculty of Natural Sciences and Technology, Department of Chemical Engineering, 2008. ISBN 9788247113554. URL http://books.google.no/books?id=5h9GMwEACAAJ.

Timothy Maly and Linda R. Petzold. Numerical methods and software for sensitivity analysis of differential-algebraic systems. *Applied Numerical Mathematics*, 20(12):57 – 79, 1996. ISSN 0168-9274. doi: 10.1016/0168-9274(95)00117-4. URL http://www.sciencedirect.com/science/article/pii/0168927495001174. ¡ce:title¿Method of Lines for Time-Dependent Problems¡/ce:title¿.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *FPCA*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer, 1991. ISBN 3-540-54396-1.

C. Mischler, X. Joulia, E. Hassold, A. Galligo, and R. Esposito. Automatic differentiation applications to computer aided process engineering. *Computers & Chemical Engineering*, 19, Supplement 1(0):779 – 784, 1995. ISSN 0098-1354. doi: 10.1016/0098-1354(95)87129-2. URL http://www.sciencedirect.com/science/article/pii/0098135495871292.

Otto. Redlich and J. N. S. Kwong. On the thermodynamics of solutions. v. an equation of state. fugacities of gaseous solutions. *Chemical Reviews*, 44(1):233–244, 1949. doi: 10.1021/cr60137a013. URL `http://pubs.acs.org/doi/abs/10.1021/cr60137a013`. PMID: 18125401.

A.S. Silva and M. Castier. Automatic differentiation and implementation of thermodynamic models using a computer algebra system. *Computers & Chemical Engineering*, 17, Supplement 1(0):S473 – S478, 1993. ISSN 0098-1354. doi: 10.1016/0098-1354(93)80268-R. URL `http://www.sciencedirect.com/science/article/pii/009813549380268R`.

E.B. Smith. *Basic chemical thermodynamics*. Imperial college Press, 2004. ISBN 9781860944468. URL `http://books.google.no/books?id=zmIwN-FRd4cC`.

Ross Taylor. Automatic derivation of thermodynamic property functions using computer algebra. *Fluid Phase Equilibria*, 129(12):37 – 47, 1997. ISSN 0378-3812. doi: 10.1016/S0378-3812(96)03184-6. URL `http://www.sciencedirect.com/science/article/pii/S0378381296031846`.

G. Valiente. *Algorithms on Trees and Graphs*. Springer, 2002. ISBN 9783540435501. URL `http://books.google.no/books?id=NSfIWxqPlbcC`.

Appendix A

MDO source code

mdo_main.rb

```
#!/usr/bin/env ruby

require './mdo_operators.rb'
#require './mdo_string.rb'

class MDO
    # The MDOop adds a dimension to an MDO
    # MDOop syntax:           a = MDO::MDO(x, y, z)
    # mathmatical equivalent: a = [x, y, z]
    class MDOop < MultidimOperator
        def initialize(dim, dep, val)
            connect([], dim, dep, val)
        end

        def symbol_string
            return 'MDO'
        end

        def to_s
            if @dim[0].respond_to?(:eval) # an indefinite size MDO
                if @dep.length>1 # this is a hybrid MDO
                    return "(MDO::MDO("+@dep[0..-2].to_s[1..-2]+") << "+
                    @dep[-1].to_s+')'
                else # pure indefinite size MDO
                    return '('+@dep[0].to_s+'.bc('+@dim[0].to_s+
                    @dim[1..-1].collect{
                        |i| ', '+0.to_s
                    }.join+'))'
                end
            else # A constant size MDO
                return "MDO::MDO("+@dep.to_s[1..-2]+")"
```

```
32            end
33        end
34
35        def dep_strings
36            return @dep.collect{'e'}
37        end
38    end
39
40    ############################
41    # MDO class workings #
42    ############################
43
44    # class variables
45    @@last_id = 0 # Used for node identification,
46    # checking if the node is a free variable and the like
47    @@id_var_hash = {} # for finding node references from id
48
49    def incr_id # This function assigns a unique ID to an MDO node
50        @@last_id = @@last_id + 1 # ensures that different MDOs get
51        # different IDs
52        @@id_var_hash[@@last_id] = self
53        return @@last_id
54    end
55
56    # accessors
57    attr_accessor :dim, :dep, :val, :label, :operator
58
59    # This method is called by writing MDO::new
60    # it returns a free variable MDO with dimensions dim_in
61    def initialize(*dim_in)
62
63        if (dim_in !=[]) # unpack the dimension if there is argument
64
65            a = MDO.new()
66
67            dim_in.reverse.each{ |dim|
68                a = MDO.infiniteMDO(a, dim) # recursively add
69                # dimensions
70            }
71
72            @dep = a.dep.dup
73            @dim = a.dim.dup
74            @id = incr_id
75            @val = a.val.dup
76            @operator = a.operator
77        else
78            @dep = []
79            @dim = []
80            @id = incr_id
81            @val = []
```

```
        end
    end

    def getID
        return @id
    end

    def setID(idin)
        @id = idin
    end

    def label!(stringin)
        @label = stringin
        return self
    end

    # the gradient method. Returns an MDO object containing the
    # expression for the gradient with respect to vars
    def grad(vars)
        if @spec_grad==nil # check if any gradient functions have
        # been specified
            # if not, then differentiate
            return diff(vars).clean!
        else
            if @spec_grad.has_key?(vars)
                return @spec_grad[vars] # return the specified
                # gradient
            else
                return diff(vars).clean! # could not find the
                # a specified gradient with the right variables
            end
        end
    end

    def grad!(vars_in, expr_in) # define the gradient using an MDO
    # vars_in are the differentiation variables
    # expr_in is the user-spcified gradient with respect to
    # vars_in
        if (@spec_grad.respond_to?(:keys))
            @spec_grad[vars_in] = expr_in
        else
            @spec_grad = {}
            @spec_grad[vars_in] = expr_in
        end
    end

    # Differetiate with respect to a
    def diff(a, outerind=[]) # outerind keeps track of how many
        # dimensions are dereferenced before the free variable
        # is reached, this is used in the kronecker operator
```

```
        # a could have 0 or 1 dimension

        if a.dim != [] # check if it has a dimension
            if  (a.dim[0].respond_to?(:eval)) # indefinite size a
                if a.dep.length>1 # hybrid a

                    outerind1 = [1.0]
                    outerind2 = []

                    first_list = a.dep[0..-2].collect{
                        |i| diff(i, outerind1)
                    } # cut out the first part of the list and
                    # append the indefinite part

                   out = MDO.finiteMDO(first_list) <<
                         diff(a.dep[-1], outerind2)
                else # "a" is a pure indefinite size MDO
                    outerind = [1.0]

                    out = MDO.infiniteMDO(diff(a.dep[0], outerind),
                          a.dim[0]) # creates an unspecified MDO
                    # with new element reference.
                end
                return out
            else # it is a pure non-lambda MDO
                struct_list = a.dep.collect{|y| diff(y, outerind)}
                return MDO.finiteMDO(struct_list)
            end
        else # by the time this is reached, a is split up into its
        # elements
            return MDO.const(1.0) if a.getID==@id # the derivative
            # of the free variable is one

            if @operator != nil # if it has an operator
                if @operator.class == MDOop # an this is MDOop
                    if  (@dim[0].respond_to?(:eval)) # indefinite size
                        if @dep.length>1 # hybrid dimension
                            outerind1 = outerind.dup << 1.0
                            outerind2 = outerind.dup
                            non_lambda = @dep[0..-2].collect{
                                |i| i.diff(a, outerind1)
                            }
                            return MDO.finiteMDO(non_lambda) <<
                                   @dep[-1].diff(a, outerind2)
                        else # it is a pure indefinite dimension
                            # check if independent var
                            if (@dep[-1].getID==a.getID) # The dimension
                                # is of a free variable vector

                                outerind1 = outerind.dup << 1.0
```

```
                                # save the dimension corresponding
                                # to the differentiation variable
                                arg2kronecker = outerind1.length

                                # create a kronecker delta MDO
                                return MDO.infiniteMDO(
                                        MDO.kronecker(arg2kronecker),
                                        @dim[0])
                            else # it is not the free variable
                                outerind1 = outerind.dup << 1.0
                                return MDO.infiniteMDO(@dep[-1].diff(a,
                                        outerind1), @dim[0])
                            end
                        end
                    else # it is a constant dimension MDO
                        outerind1 = outerind.dup << 1.0

                        return MDO.finiteMDO( @dep.collect{|x|
                             x.diff(a, outerind1)
                             } )
                    end
                else # has an operator, but it is not MDOop
                    return @operator.diff(a, outerind)
                end
            else
                return MDO.const(0.0)
            end
        end
    end

    def array_dig(var_values_in, dim_in, num) # organize
        # evaluation results in a nested list
        dim_f = dim_in.dup
        if num < dim_in.length
            if dim_f[num].respond_to?(:eval)
                dim_f[num] = dim_f[num].eval(var_values_in, [])
                # evaluate MDO for dimension
            end

            raise "unspecified dimension" if dim_f.include?(nil)

            return Array.new(dim_f[num]) {|i|
                 dim_f[num] = i; array_dig(var_values_in, dim_f.dup, num+1)
                 }
        else
            return self.eval(var_values_in, dim_in)
        end
    end

```

```
232     def eval(var_values = {}, index=:all, abs_index=[])
233         # var_values is a hash table associating variables
234         # with their values
235 
236         if (!var_values.has_key?(:not_top)) # if the var_values are
237         #given by using MDO references as keys ->
238         #Switch to using the id's as keys
239             temp_hash = {}
240             var_values.each{ |key, value|
241                 if value.respond_to?(:each) and
242                 key.dim[0].respond_to?(:eval) # one of the arguments
243                 # to the eval function is a list, and its variable has
244                 # an unspecified dimensions size
245 
246                     if !var_values.has_key?(key.dim[0])
247                         # hasn't got specified dim
248                         if !(key.dim[0].dep[0].class == Fixnum)
249                            # The dimension is not a constant
250                             temp_hash[key.dim[0]] = value.length
251                         else # the dimension is specified constant
252                             spec_dim = key.dim[0].eval()
253 
254                             if !(value.length == spec_dim)
255                                 puts "warning: specified dimension mismatch
    , predefined constant length = #{spec_dim}, while supplied free
    variable length = #{value.length}. Using predifened dimension"
256                                 temp_hash[key.dim[0]] = spec_dim
257                             else
258                                 temp_hash[key.dim[0]] = value.length
259                             end
260                         end
261                     else
262                         if (var_values[key.dim[0]] != value.length)
263                             puts "warning: specified dimension mismatch,
    dimension length supplied upon evaluation = #{var_values[key.dim[0]]},
    while supplied free variable length = #{value.length}. Using specified
    dimension, not free variable length"
264                         end
265                     end
266                 end
267             }
268             temp_hash.each{|key, value| var_values[key] = value} # add
269             # the newly found dimensions
270 
271             # This next bit relates to higher rank free variables
272             unwarped = {}
273 
274             var_values.each{ |key, value|
275                 if value.respond_to?(:length)
276                     if value[0].respond_to?(:length) # nested list
```

```
                        # if it is of rank 2 or above,
                        # warp it into a constant for the course
                        # of the evaluation
                        unwarped[key] = MDO::copy(key);
                        key.copy_constructor!(MDO::express(value))
                    end
                end
            }

            # the warped variable, are no longer free variables
            # but redefined as "contants"
            unwarped.each{|key, value| var_values.delete(key)}

            var_values_id = {}
            var_values.each{|key, value| var_values_id[key.getID] = value}
            var_values = var_values_id

            var_values[:not_top] = true # key to the var_values argument
            # to indicate that further eval calls are not the top call
            # evaluation can therefore proceed as it was intended
            # from this point

            outvalue = eval(var_values, index, abs_index)

            # revert rank 2 or above independent variables
            # from constants to variables
            unwarped.each{|key, value| key.copy_constructor!(value)}

            return outvalue
        end

        # It will only get here if it is not the top eval call
        # this is however, where all the evaluation takes place
        if index == :all
            # call eval with all index combinations and put them in a list
            if @dim.first!=nil # test if the dimensionality is an integer
                out = array_dig(var_values, @dim.dup, 0) # nested list
            else
                out = self.eval(var_values, [])
            end
            return out
        else
            if var_values.has_key?(getID)
                if var_values[getID].respond_to?(:/) # if it is not
                #an array
                    return var_values[getID]
                else # it is an array
                    return var_values[getID][index[0]]
                end
            end
```

```
327
328             if (@operator != nil)
329                 if @operator.class == MDOop
330                     if (!dim[0].respond_to?(:eval)) # constant size MDO
331
332                         shaved = index[0]
333                         remainder = index[1..-1]
334                         abs_index_down = abs_index.dup << index[0]
335
336                         return @dep[shaved].eval(var_values,
337                                 remainder, abs_index_down)
338                     end
339                     if (dim[0].respond_to?(:eval) and @dep.length>1)
340                     # hybrid
341                         if (index[0]<(@dep.length-1-0.1))
342                         # in the constant size section
343
344                             shaved = index[0]
345                             remainder = index[1..-1]
346                             abs_index_down = abs_index.dup << index[0]
347
348                             return @dep[shaved].eval(var_values,
349                                         remainder, abs_index_down)
350                         else
351                         # in the indefinite section
352
353                             index_down = index
354                             index_down[0] = (index_down[0]-(@dep.length-1))
355
356                             return @dep[-1].eval(var_values, index_down,
357                                             abs_index)
358                         end
359                     end
360                     if (dim[0].respond_to?(:eval) and @dep.length==1)
361                     # pure indefinite size MDO
362                         if (var_values.has_key?(@dep[-1].getID))
363                             # have a free variable MDO
364                             return @dep[-1].eval(var_values, index,
365                                         abs_index)
366                         end
367                         # if not, then delete the top index and keep going
368                         remainder = index[1..-1]
369                         abs_index_down = abs_index.dup << index[0]
370
371                         return @dep[-1].eval(var_values, remainder,
372                                         abs_index_down)
373                     end
374                 else
375                     return @operator.eval(var_values, index, abs_index)
376                 end
```

```
377             else
378                 return @val if @val != nil # We have a constant
379             end
380
381             # if this part is reached, then a variable has been
382             # left unspecified. eval will return nil
383
384             puts "Warning: reached unspecified variable (id = #{getID},
    label = #{@label}), make sure that all necessary variables are
    specified when calling eval. returning nil"
385             return nil
386         end
387     end
388
389     # constant constructor
390     def self.const(a)
391         c = MDO.new
392         c.val = a
393         c.dim = []
394         return c
395     end
396
397     # return the rank
398     def rank
399         return @dim.length
400     end
401
402     # produce a indefinite size MDO
403     def self.infiniteMDO(lambda_func, dimension)
404         c = MDO.new
405         c.operator = MDOop::new(c.dim, c.dep, c.val)
406         c.dep[0..-1] = [lambda_func]
407
408         c.dim[0..-1] = (!dimension.respond_to?(:eval) ? (dimension==0? (
    raise "tried to add an MDO dimension of length 0") : [MDO.const(
    dimension)]) : [dimension]) + lambda_func.dim
409
410         return c
411     end
412
413     # produce a constant size MDO
414     def self.finiteMDO(*elements)
415         structure = elements.flatten
416
417         c = MDO.new
418         c.operator = MDOop::new(c.dim, c.dep, c.val)
419         c.dep[0..-1] = structure
420
421         dims = structure.collect{|i| i.dim}
422         larg_dim = []
```

```
        dims.each{|d| larg_dim = d if d.length>larg_dim.length}
        c.dim[0..-1] = [structure.length] + larg_dim
        return c
    end

    # constructor used for creating a constant size MDO
    def self.MDO(*structure)
        flattened = structure.flatten

        return MDO.finiteMDO(flattened)
    end

    def assim!(lambda_in)
        # lambda_in is a function with an MDO in dim[0]
        # the object calling assim has to have operator.class == MDOop
        if (@operator.class==MDOop)
            @dep << lambda_in
            old_dim = @dim[0]

            if !lambda_in.dim[0].respond_to?(:eval)
                raise "lambda dimension needs to be bound to dimension object"
            end
            @dim[0] = MDO.const(old_dim).add(lambda_in.dim[0])
            return self
        else
            raise "type error, only operator.class == MDOop MDOs can assimilate"
        end
    end

    def <<(y)
        ymod = MDO.express(y)
        assim!(ymod)
    end

    # the structuring from list function
    def self.express(input)

        # MDO
        return input if input.respond_to?(:eval)

        # NUMBER
        return MDO.const(input) if !input.respond_to?(:length)

        # ARRAY
        return MDO.finiteMDO( input.collect{|i| express(i)} )
    end

    # To allow for 4+x as well as x+4
```

```
    def coerce(other)
        return MDO.express(other), self
    end
end

#require './mdo_dot.rb'
require './mdo_clean.rb'
# require './maple_simplified.rb'

Dim  = MDO
Index = MDO
```

source/mdo_main.rb

mdo_operators.rb

```
#!/usr/bin/env ruby


class MDO
# @operator = nil # a constant or a free variable

    ##########################################################
    #  PARENT CLASS FOR ALL SIMPLE/SCALAR/LAMBDA OPERATORS #
    ##########################################################

    class LambdaOperator
        attr_accessor :symbol_string, :dep_strings, :dep, :dim, :val
        def connect(a, dim, dep, val) # connect nodes in graph
            @dim = dim
            @dep = dep
            @val = val

            @dep[0...a.length] = a

            larg_index = 0
            dim_lengths = @dep.collect{|d| d.dim.length}
            dim_lengths.each_index{|i|
                    if dim_lengths[i]>=dim_lengths[larg_index]
                        larg_index = i
                    end
                }

            @dim[0..-1] = @dep[larg_index].dim
        end
    end

            ####################
            #  MULTIPLICATION #
            ####################
```

```
35
36          class Mult < LambdaOperator
37              def initialize(a, b, dim, dep, val)
38                  @symbol_string = "*"
39                  @dep_strings = ['f', 'f']
40                  connect([a, b], dim, dep, val)
41              end
42
43              def to_s
44                  return '('+@dep[0].to_s+@symbol_string+@dep[1].to_s+')'
45              end
46
47              def diff(a, outerind)
48                  return @dep[0].mult(@dep[1].diff(a, outerind))
49                  .add(@dep[1].mult(@dep[0].diff(a, outerind)))
50              end
51
52              def eval(var_values, index, abs_index)
53                  return @dep[0].eval(var_values, index, abs_index) *
54                  @dep[1].eval(var_values, index, abs_index)
55              end
56          end
57
58          def mult!(a, b)
59              @operator = Mult::new(a, b, @dim, @dep, @val)
60          end
61
62          def mult(b)
63              c = MDO.new
64              c.mult!(self, b)
65              return c
66          end
67
68          def *(y)
69              ymod = MDO.express(y)
70              return mult(ymod)
71          end
72
73          #################
74          #  ADDITION #
75          #################
76
77          class Add < LambdaOperator
78              def initialize(a, b, dim, dep, val)
79                  @symbol_string = "+"
80                  @dep_strings = ['t', 't']
81                  connect([a, b], dim, dep, val)
82              end
83
84              def to_s
```

```
                    return '('+@dep[0].to_s+@symbol_string+@dep[1].to_s+')'
                end

                def diff(a, outerind)
                    return @dep[0].diff(a, outerind).add(@dep[1].diff(a,
                                 outerind))
                end

                def eval(var_values, index, abs_index)
                    return @dep[0].eval(var_values, index, abs_index) +
                           @dep[1].eval(var_values, index, abs_index)
                end
            end

            def add(b)
                c = MDO.new
                c.operator = Add::new(self, b, c.dim, c.dep, c.val)
                return c
            end

            def +(y)

                ymod = MDO.express(y) # should call this function here, to
    make y into
                # an MDO from whatever input
                return add(ymod)
            end

            #################
            #  SUBTRACTION #
            #################

            class Sub < LambdaOperator
                def initialize(a, b, dim, dep, val)
                    @symbol_string = "-"
                    @dep_strings = ['m', 's']
                    connect([a, b], dim, dep, val)
                end

                def to_s
                    return '('+@dep[0].to_s+@symbol_string+@dep[1].to_s+')'
                end

                def diff(a, outerind)
                    return @dep[0].diff(a, outerind).sub(@dep[1].diff(a,
                            outerind))
                end

                def eval(var_values, index, abs_index)
                    return @dep[0].eval(var_values, index, abs_index) -
```

```
                        @dep[1].eval(var_values, index, abs_index)
                    end
                end

                def sub(b)
                    c = MDO.new
                    c.operator = Sub::new(self, b, c.dim, c.dep, c.val)
                    return c
                end

                def -(y) # Subtraction
                    ymod = MDO.express(y)
                    return sub(ymod)
                end

                def -@ # Negative sign
                    return -1.0*self
                end

                #################
                #  DIVISION     #
                #################

                class Div < LambdaOperator
                    def initialize(a, b, dim, dep, val)
                        @symbol_string = "/"
                        @dep_strings = ['n', 'd']
                        connect([a, b], dim, dep, val)
                    end

                    def to_s
                        return '('+@dep[0].to_s+@symbol_string+@dep[1].to_s+')'
                    end


                    def diff(a, outerind)
                        return (@dep[0].diff(a, outerind).div(@dep[1]))
                        .sub(@dep[1].diff(a, outerind).mult(@dep[0]
                        .div(@dep[1].pow(MDO.const(2.0)))))
                    end

                    def eval(var_values, index, abs_index)
                        return @dep[0].eval(var_values, index, abs_index) /
                         @dep[1].eval(var_values, index, abs_index)
                    end
                end

                def div(b)
                    c = MDO.new
                    c.operator = Div::new(self, b, c.dim, c.dep, c.val)
```

```
184                 return c
185             end
186
187             def /(y)
188                 ymod = MDO.express(y)
189                 return div(ymod)
190             end
191
192             #######################
193             #  NATURAL LOGARITHM #
194             #######################
195
196             class Ln < LambdaOperator
197                 def initialize(a, dim, dep, val)
198                     @symbol_string = "ln"
199                     @dep_strings = ['a']
200                     connect([a], dim, dep, val)
201                 end
202
203                 def to_s
204                     return '('+@dep[0].to_s+'.'+@symbol_string+')'
205                 end
206
207                 def diff(a, outerind)
208                     return @dep[0].diff(a, outerind).div(@dep[0])
209                 end
210
211                 def eval(var_values, index, abs_index)
212                     return Math.log(@dep[0].eval(var_values, index,
213                      abs_index))
214                 end
215             end
216
217             def ln
218                 c = MDO.new
219                 c.operator = Ln::new(self, c.dim, c.dep, c.val)
220                 return c
221             end
222
223             #################
224             #  EXPONENTIAL  #
225             #################
226
227             class Exp < LambdaOperator
228                 def initialize(a, dim, dep, val)
229                     @symbol_string = "exp"
230                     @dep_strings = ['a']
231                     connect([a], dim, dep, val)
232                 end
233
```

```
                def to_s
                    return '('+@dep[0].to_s+'.'+@symbol_string+')'
                end

                def diff(a, outerind)
                    return @dep[0].diff(a, outerind).mult(@dep[0].exp)
                end

                def eval(var_values, index, abs_index)
                    return Math.exp(@dep[0].eval(var_values, index, abs_index))
                end
            end


            def exp
                c = MDO.new
                c.operator = Exp::new(self, c.dim, c.dep, c.val)
                return c
            end

            ######################
            #  POWERS/EXPONENTS #
            ######################

            class Pow < LambdaOperator
                def initialize(a, b, dim, dep, val)
                    @symbol_string = "**"
                    @dep_strings = ['b', 'e']
                    connect([a, b], dim, dep, val)
                end

                def to_s
                    return '('+@dep[0].to_s+@symbol_string+@dep[1].to_s+')'
                end

                def diff(a, outerind)
                    return @dep[1].mult(@dep[0].pow(@dep[1].sub(
                        MDO.const(1.0))))
                        .mult(@dep[0].diff(a, outerind))
                end

                def eval(var_values, index, abs_index)
                    return  @dep[0].eval(var_values, index, abs_index) **
                            @dep[1].eval(var_values, index, abs_index)
                end
            end

            def pow(b)
                c = MDO.new
```

```
                c.operator = Pow::new(self, b, c.dim, c.dep, c.val)
                return c
            end

            def **(y)
                ymod = MDO.express(y)
                return pow(ymod)
            end

    ####################################################
    #  PARENT CLASS FOR ALL MULTIDIMENTIONAL OPERATORS #
    ####################################################

    class MultidimOperator
        attr_accessor :dep, :dim, :val
        def connect(a, dim, dep, val)
            @dim = dim
            @dep = dep
            @val = val

            @dep[0...a.length] = a # dup ?
        end

    end

    def deref_dim(var_values = {})
        a = @dim.collect{|i| i.respond_to?(:eval) ? i.eval(var_values,
        []) : i}
        return a
    end # function used to check and evaluate if a dimensions is
    # represented by a dimension variable

            #################
            #  INDEXING     #
            #################

            class Ind < MultidimOperator # The indexing operator class

                def initialize(a, b, dim, dep, val, col_dim)
                    connect([a, b], dim, dep, val)

                    @val[0] = col_dim

                    g = @dep[0].dim.dup
                    g.delete_at(col_dim)
                    @dim[0..-1] = g.dup
                end

                def symbol_string
                    return '['+@val[-1].to_s+']'
```

```
333                 end
334
335                 def dep_strings
336                     return ['a', 'i']
337                 end
338
339                 def to_s
340                     return '('+@dep[0].to_s+'['+@dep[1].to_s+', '+
341                             @val[-1].to_s+']'+')'
342                 end
343
344                 def diff(a, outerind)
345                     return  @dep[0].diff(a, outerind).index(@dep[1],
346                             @val[-1])
347                 end
348
349                 def eval(var_values, index, abs_index)
350                      index_down = index.insert(@val[-1],
351                                   @dep[1].eval(var_values))
352
353                      return @dep[0].eval(var_values, index_down, abs_index)
354                 end
355             end
356
357             def index(index_in, col_dim)
358                 c = MDO.new
359                 c.operator = Ind::new(self, index_in, c.dim, c.dep,
360                                       c.val, col_dim)
361
362                 return c
363             end
364
365             def [](y, dimen=0)
366                 return index(MDO.express(y), dimen)
367             end
368
369             #################
370             #  SUMMATION    #
371             #################
372
373             class Sum < MultidimOperator
374                 def initialize(a, dim, dep, val, col_dim)
375
376                     connect([a], dim, dep, val)
377
378                     @dep[0] = a
379                     @val[0] = col_dim
380
381                     g = @dep[0].dim.dup
382                     g.delete_at(col_dim)
```

```
                    @dim[0..-1] = g.dup
                end

                def symbol_string
                    return 'sum('+@val[-1].to_s+')'
                end

                def dep_strings
                    return ['a']
                end

                def to_s
                    return '('+@dep[0].to_s+'.'+symbol_string+')'
                end

                def diff(a, outerind)
                    return  (dep[0].diff(a, outerind)).sum(@val[-1])
                    # the sum of the derivatives
                end

                def eval(var_values, index, abs_index)
                    sum = 0.0
                    (0...@dep[0].deref_dim(var_values)[@val[-1]])
                    .collect{ |k|
                        @dep[0].eval(var_values,
                                     index.dup.insert(@val[-1], k),
                                     abs_index.dup)
                        }.each{ |i| sum = sum + i }
                    return sum
                end
            end

            def sum(col_dim=0) # collapsing dimension
                c = MDO.new
                c.operator = Sum::new(self, c.dim, c.dep, c.val, col_dim)

                return c
            end

            #################
            #  BROADCASTING #
            #################

            class Broadcast < MultidimOperator # index (verb). Not to be confused with Index (noun) (which is a special type of Expr)
                def initialize(a, dim, dep, val, spec)

                    connect([a], dim, dep, val)

                    @dep[0] = a
```

```
                    @val[0..-1] = spec.dup

                    c = -1
                    @dim[0..-1] = spec.collect{|i| i==0? (c=c+1; @dep[0].
    dim[c]) : i }

                end

                def symbol_string
                    return 'bc('+@val.collect{|i| i.to_s}.join(', ')+')'
                end

                def dep_strings
                    return ['l']
                end

                def to_s
                    return '('+@dep[0].to_s+'.'+symbol_string+')'
                end

                def diff(a, outerind)

                    outerind1 = outerind.dup + @val.select{|i|
                                        i!=0}.collect{|j| 1.0}

                    return dep[0].diff(a, outerind1).broadcast(@val)
                end

                def eval(var_values, index, abs_index)
                    # index coming in [1, 1]
                    # if broadcast is [0, 1] => send down indices
                    # corresponding to the positions of the ones!
                    # example index = [1, 3, 5, 2, 0]
                    # example broad = [0, 1, 0, 0, 1]
                    # sent down     = [   3,       0] = [3, 0]

                    filter = @val.collect{|i| i==0? 1 : 0}

                    index_down = index.zip(filter).select{|i|
                        (i[1]!=0 and i[1]!=nil)}.collect{|i| i[0]}

                    index_popped = index.zip(filter).select{|i|
                        (i[1]==0 or i[1]==nil)}.collect{|i| i[0]}

                    abs_index_down = abs_index + index_popped

                    return @dep[0].eval(var_values, index_down,
                           abs_index_down)
                end
            end
```

```
481
482             # Broadcasting specification:
483
484             #    a.dim = [c, c]
485             #
486             #    a.bc(c, 0, 5, 0) # produces a broadcast of
487             #    dim = [c, c, 5, c]
488             #    0 or nil means do not broadcast in that
489             #    direction (in other words:
490             #    keep the indices)
491
492             def broadcast(spec)
493
494                 c = MDO.new
495                 c.operator = Broadcast::new(self, c.dim, c.dep, c.val,
496                                             spec)
497
498                 return c # to be changed
499             end
500
501             def bc(*args)
502                 broadcast(args.flatten)
503             end
504
505             def call(*args)
506                 broadcast(args.flatten)
507             end
508
509             #################
510             #  KRONECKER    #
511             #################
512
513             class KroneckerDelta < MultidimOperator
514                 def initialize(dim, dep, val, outerind)
515                     connect([], dim, dep, val)
516
517                     @val << outerind
518                 end
519
520                 def symbol_string
521                     return 'kronecker('+@val.collect{|i| i.to_s}
522                             .join(', ')+')'
523                 end
524
525                 def dep_strings
526                     return []
527                 end
528
529                 def to_s
530                     return 'MDO::kronecker('+@val.collect{|i| i.to_s}
```

```
                        .join(', ')+')'
            end

            def diff(a, outerind)
                return MDO.const(0.0)
            end

            def eval(var_values, index, abs_index)
                abs_index_mod = abs_index[-@val[-1]...-@val[-1]+1] +
                                [abs_index[-1]]


                # compares the specified index number @val,
                # with the final remaining index
                # the value of @val is not intuitive and is
                # specified during
                # grad. It is hardly meant to be used manually

                out = (abs_index_mod.uniq.length == 1 or
                            abs_index_mod.length == 0)? 1.0 : 0.0
                return out
            end
        end

        def self.kronecker(outerind=0)

            c = MDO.new
            c.operator = KroneckerDelta::new(c.dim, c.dep, c.val,
                                             outerind)

            return c
        end

        #################
        #  PERMUTATION  #
        #################

        class Permute < MultidimOperator
            def initialize(a, dim, dep, val, spec)

                connect([a], dim, dep, val)

                @dep[0] = a
                @val[0..-1] = spec.dup

                @dim[0..-1] = a.dim.dup

                # check if the number if specified arguments are valid
                if !(spec.sort == (0...dim.length)
                                  .collect{|i| i})
```

```
                    raise 'invalid permutation: for a 2d MDO: (0, 1) and (1, 0) valid, for a 3d MDO (0, 1, 2), (0, 2, 1), (1, 0, 2), (1, 2, 0), (2, 1, 0) and (2, 0, 1) allowed and so on...'
                end
            end

            def symbol_string
                return 'perm('+@val.collect{|i| i.to_s}.join(', ')+')'
            end

            def dep_strings
                return ['a']
            end

            def to_s
                return '('+@dep[0].to_s+'.'+symbol_string+')'
            end

            def diff(a, outerind)
                outerind1 = outerind.dup # this does not change,
                # as no indices are used for dereferencing
                # in this operator
                return dep[0].diff(a, outerind1).permute(@val)
            end

            def eval(var_values, index, abs_index)

                index_down = @val.collect{|i| index[i]}

                abs_index_down = abs_index

                return @dep[0].eval(var_values, index_down,
                                    abs_index_down)
            end
        end

        def permute(spec) # spec is a list of arguments

            c = MDO.new
            c.operator = Permute::new(self, c.dim, c.dep, c.val, spec)

            return c
        end

        def per(*args) # this is supposed to be the user's
                       # way to permute

            permute(args.flatten)
        end
```

```ruby

end
```

source/mdo_operators.rb

mdo_clean.rb

```ruby
#!/usr/bin/env ruby

class MDO

    # ideas for cleaning:
    # sum -> struct -> 0.0 THIS ONE
    # ^1.0, 0.0

    def self.copy(input)
        c = MDO.new
        c.copy_constructor!(input)
        return c
    end

    def copy_constructor!(expr_in)
        @dim = expr_in.dim.dup
        @val = expr_in.val
        @label = (expr_in.label.class!=NilClass)?
                  expr_in.label.dup : expr_in.label
        @dep = expr_in.dep.dup
        @id = expr_in.getID
        @operator = (expr_in.operator.class!=NilClass)?
                     expr_in.operator.dup : expr_in.operator
    end

    def clean!
        still_changing = true
        while still_changing
            still_changing = self_sweep!
        end
        return self
    end

    def self_sweep!()
        changed = false

        # if all dependents are constants: Evaluate
        if @operator.class < LambdaOperator
            if (@dep.all?{|d| (d.operator==nil and
                                d.val.respond_to?(:/))})
                temp = MDO::const(eval)
                copy_constructor!(temp)
                changed = true
```

```
44              end
45          end
46          if @operator.class == Add # Realized in hindsight that case
47          # would be better at this, but has to be used as case @operator
48          # when Add etc. not case @operator.class when Add
49              if (@dep[0].operator==nil and @dep[0].val==0.0)
50                  if (@dep[1].operator==nil and @dep[1].val==0.0)
51                      copy_constructor!(MDO::const(0.0))
52                      changed = true
53                  else
54                      copy_constructor!(@dep[1])
55                      changed = true
56                  end
57              else
58                  if (@dep[1].operator==nil and @dep[1].val==0.0)
59                      copy_constructor!(@dep[0])
60                      changed = true
61                  end
62              end
63          end
64          if @operator.class == Sub
65              if (@dep[0].operator==nil and @dep[0].val==0.0)
66                  if (@dep[1].operator==nil and @dep[1].val==0.0)
67                  # both 0 ok, return 0
68                      copy_constructor!(MDO::const(0.0))
69                      changed = true
70                  else
71                      mult!(MDO.const(-1.0), dep[1])
72                      # only the minuend 0, return -1.0*dep[1]
73                      changed = true
74                  end
75              else
76                  if (@dep[1].operator==nil and @dep[1].val==0.0)
77                      copy_constructor!(@dep[0])
78                      changed = true
79                  end
80              end
81          end
82          if @operator.class == Mult
83              if ((@dep[0].operator==nil and @dep[0].val==0.0) or
84                  (@dep[1].operator==nil and @dep[1].val==0.0))
85                  copy_constructor!(MDO::const(0.0))
86                  changed = true
87              else
88                  if (@dep[0].operator==nil and @dep[0].val==1.0)
89                      copy_constructor!(@dep[1])
90                      changed = true
91                  else
92                      if (@dep[1].operator==nil and @dep[1].val==1.0)
93                          copy_constructor!(@dep[0])
```

```
                        changed = true
                    end
                end
            end
        end
        if @operator.class == Div
            if (@dep[0].operator==nil and @dep[0].val==0.0)
            # if numerator zero, whole zero
                copy_constructor!(MDO::const(0.0))
                changed = true
            else
                if (@dep[1].operator==nil and @dep[1].val==1.0)
                # if the denominator is one, then numerator
                    copy_constructor!(@dep[0])
                    changed = true
                end
            end
        end
        if @operator.class == Ln
            if (@dep[0].operator==nil and @dep[0].val==1.0)
                copy_constructor!(MDO::const(0.0))
                changed = true
            end
        end
        if @operator.class == Exp
            if (@dep[0].operator==nil and @dep[0].val==0.0)
                copy_constructor!(MDO::const(1.0))
                changed = true
            end
        end
        if @operator.class == Pow
            if (@dep[1].operator==nil and @dep[1].val==1.0)
                copy_constructor!(@dep[0])
                changed = true
            else
                if (@dep[0].operator==nil and @dep[0].val==0.0)
                    copy_constructor!(MDO::const(0.0))
                    changed = true
                end
            end
        end

        if @operator.class == Sum # sum of zero is zero
            if (@dep[0].operator.class==MDOop and
                 @dep[0].dep[0].operator==nil and
                 @dep[0].dep[0].val==0.0)  # sum of a 0 vector
                copy_constructor!(MDO::const(0.0))
                changed = true
            end
        end
```

```
        @dep.each{|i| (if changed
                            i.self_sweep!;
                        else;
                            changed = i.self_sweep!;
                        end) if i.respond_to?(:self_sweep!)}
        return changed
    end

end
```

source/mdo_clean.rb

Appendix B

Thermodynamic model implementation

mdo_impl_ex.rb

```
#!/usr/bin/env ruby

require './mdo_main.rb'

##############################################
# BUILDING THE MODEL USING EXPRESSION OBJECTS #
##############################################

# CONSTANTS AND VARIABLES
######################
r = MDO::const(8.314)
c = Dim::new # the number of components defined as a dimension

t = MDO::new # temperature
v = MDO::new # volume

n_vec = MDO::new(c) # mole number vector

tc_vec = MDO::new(c) # critical temperature vector
pc_vec = MDO::new(c) # critical pressure vector

# RK-EOS SPECIFIC
######################
b_vec = 0.0867*r*tc_vec/pc_vec # vector of b values for each component
a_vec = 0.4278*r**2.0*tc_vec**2.5/pc_vec # vector of a values for each
    component

k_mat = MDO::new(c, c) # interaction parameter matrix
```

```
28
29
30 a_mat = (a_vec.bc(0, c)*a_vec.bc(c, 0))**0.5*(1.0-k_mat)
31
32 b = (n_vec*b_vec).sum
33 a = (n_vec.bc(0, c)*n_vec.bc(c, 0)*a_mat).sum.sum
34
35 n_tot = n_vec.sum
36
37 p_rk = n_tot*r*t/(v-b) - a/(t**0.5*v*(v+b)) # pressure = f(T,V,n) according
      to RK-EOS
38
39 f_res = n_tot*r*t*(v/(v-b)).ln+a/(b*t**0.5)*(v/(v+b)).ln # residual
     helmholtz free energy
40
41 # IDEAL GAS SPECIFIC
42 #######################
43 # Using non-constant heat capacities
44 a_cp = MDO::new(c)
45 b_cp = MDO::new(c)
46 c_cp = MDO::new(c)
47 d_cp = MDO::new(c)
48 e_cp = MDO::new(c)
49 h0 = MDO::new(c)
50 s0 = MDO::new(c)
51
52 t0 = MDO::const(298.0) # K
53 p0 = MDO::const(1.0e5) # 1 bar in Pa
54
55 #  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
56 # integral of cp(T) dT
57 int1 =  a_cp*(t-t0) +
58         b_cp*(t**2.0-t0**2.0)/2.0 +
59         c_cp*(t**3.0-t0**3.0)/3.0 +
60         d_cp*(t**4.0-t0**4.0)/4.0 +
61         e_cp*(t**5.0-t0**5.0)/5.0
62
63 # integral of cp(T)/T dT
64 int2 =  a_cp*(t/t0).ln +
65         b_cp*(t-t0) +
66         c_cp*(t**2.0-t0**2.0)/2.0 +
67         d_cp*(t**3.0-t0**3.0)/3.0 +
68         e_cp*(t**4.0-t0**4.0)/4.0
69
70
71 h = h0 + int1
72 s = s0 + int2
73 mu0_vec = h - t*s
74 mu_vec = mu0_vec + r*t*(n_vec*r*t/(v*p0)).ln
75
```

```
f_ig = -n_tot*r*t + (mu_vec*n_vec).sum

# TOTAL HELMHOLTZ FREE ENERGY
#########################

f = f_ig + f_res

x = MDO::MDO(v) << n_vec

############################
# FOR USE IN ITERATION     #
############################

y = f.grad(x)
dydx = y.grad(x)

#########################
# MODEL EVALUATION DATA #
#########################



# System 2: My arbitrary test system
####################

#               CH4          CH3CH3           CH3CH2CH3
# Critical temperature and pressure
# found from Lange's Handbook of Chemistry:
# http://books.google.no/books?id=ln0eAQAAIAAJ
tc_vals = [ 190.6   ,    305.3   ,   369.552    ] # Kelvin
pc_vals = [ 46.1e5  ,    49.0e5  ,   42.4924e5  ] # Pascal

k_vals = [  [0.0,           0.0,            0.0], # interaction
            [0.0,           0.0,            0.0], # parameter values
            [0.0,           0.0,            0.0]   ] # all set to zero

# cp parameters from 300-1000K
#  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
# Adapted from:
# http://www.me.berkeley.edu/gri\_mech/version30/files30/thermo30.dat
a_cp_vals =  [  42.8161,   35.6789      ,   7.76157  ] # J/(K mol)
b_cp_vals =  [  -0.113661 ,   -4.573982e-2 ,   0.219694  ] # J/(K^2 mol)
c_cp_vals =  [  4.08883e-4,   4.983730e-4  ,   5.07651e-5  ] # J/(K^3 mol)
d_cp_vals =  [  -4.0301535e-7 ,   -5.890189e-7 ,   -1.827209e-7  ] # J/(K^4
     mol)
e_cp_vals =  [  1.38589e-10 ,   2.2338535e-10,   7.91070889e-11  ] # J/(K^5
     mol)

# standard enthalpy and
# entropy found from:
```

```
# http://webbook.nist.gov/chemistry/
h0_vals =  [ -78870.0   ,  -84000    ,  -104700 ] # J/mol 298K (g) 1 bar
s0_vals =  [ 186.25,   229.6   ,   269.91 ] # J/(K mol) 1 bar 298K (g)

params = {tc_vec=>tc_vals, pc_vec=>pc_vals,
         a_cp => a_cp_vals,
         b_cp => b_cp_vals,
         c_cp => c_cp_vals,
         d_cp => d_cp_vals,
         e_cp => e_cp_vals,
         h0=>h0_vals, s0=>s0_vals,
         k_mat=>k_vals}

#########################
# EXAMPLE EVALUATION    #
#########################
t_val = 280.0 # K
v_val = 0.07 # m3
n_vals = [ 80.0    ,    60.0    ,   40.0        ] # moles

p y.eval(params.merge(t=>t_val, v=>v_val, n_vec=> n_vals))
p dydx.eval(params.merge(t=>t_val, v=>v_val, n_vec=> n_vals))
```

source/mdo_impl_ex.rb

Appendix C

Phase calculation iteration

numphases.rb

```
#!/usr/bin/env ruby

require 'matrix.rb'

# Prepare iteration scheme
def numphases(t, t_val, v, v_val, n_vec, n_vals, all_but_x, b, y, dydx,
              guess = nil, points = nil)

    puts "calculating if 2 or 1 phase at #{t_val}, #{v_val}"
    puts "n_vals = #{n_vals.inspect} and guess = #{guess.inspect}"
    # the above is outputted in order to be able to reproduce a
    # call in case of errors

    all_but_x.merge!(t=>t_val)

    if guess == nil or guess == []
        n_g = [n_vals[0]*0.6, n_vals[1]*0.5, n_vals[2]*0.4]
        n_l = [n_vals[0]*0.4, n_vals[1]*0.5, n_vals[2]*0.6]

        b_l = b.eval(all_but_x.merge(n_vec=>n_l))
        v_l = 1.5*b_l
        v_g = v_val-v_l
    else
        # use the guess = x_g_guess = [v_l, n_l_1, n_l_2, ...etc]
        puts "using guess: #{guess.inspect}"

        v_l = guess[0]
        v_g = v_val - v_l

        n_l = guess[1..-1] # The total mole number doesn't change
        n_g = n_vals.zip(n_l).collect{|n_val, n_l_val| n_val - n_l_val }
```

```
32
33          if (!n_g.all? {|n_g_i| n_g_i > 0.0} or
34          !n_l.all? {|n_l_i| n_l_i > 0.0}) # check if unphysical
35              puts "bad guess for n"
36              n_g = [n_vals[0]*0.6, n_vals[1]*0.5, n_vals[2]*0.4]
37              n_l = [n_vals[0]*0.4, n_vals[1]*0.5, n_vals[2]*0.6]
38          end
39
40          b_l = b.eval(all_but_x.merge(n_vec=>n_l)) # hard sphere volume
41          if v_l < b_l
42              puts "bad guess for v"
43              v_l = 1.5*b_l
44              v_g = v_val-v_l
45          end
46      end
47
48
49      x_g = [v_g] + n_g
50      x_l = [v_l] + n_l
51
52      res_norms = []
53
54      # check if gas and liquid can both be above the hard sphere volume
55      b_l = b.eval(all_but_x.merge(n_vec=>n_l))
56      b_g = b.eval(all_but_x.merge(n_vec=>n_g))
57
58      #    puts "safetyrat g = #{v_g/b_g}"
59      #    puts "safetyrat l = #{v_l/b_l}"
60
61      # Iteration loop
62      ii = -1 # for counting iterations
63      finished = false # termination criteria
64      max_iter = 40 # maximum iterations
65      tol = 1e-4 # tolerance in residual before it is "converged"
66      converged = false # has it converged?
67
68      if x_g[0]<b_g or x_l[0]<b_l # if it starts out with volumes
69          finished = true # below hard sphere, then there is no hope
70      end
71
72      while not finished
73          ii = ii + 1
74
75          y_g = Matrix.column_vector(y.eval(
76              all_but_x.merge({v=>x_g[0], n_vec=>x_g[1..-1]})))
77
78          y_l = Matrix.column_vector(y.eval(
79              all_but_x.merge({v=>x_l[0], n_vec=>x_l[1..-1]})))
80
81          dydx_g = Matrix.rows(dydx.eval(
```

```
            all_but_x.merge({v=>x_g[0], n_vec=>x_g[1..-1]}))

        dydx_l = Matrix.rows(dydx.eval(
            all_but_x.merge({v=>x_l[0], n_vec=>x_l[1..-1]}))

        jacobian = dydx_g + dydx_l

        if jacobian.singular? # this iteration scheme has failed
            res_norms << 100.0 # add an element to res_norm to
            # avoid a crash later
            break
        end

        del_x_g = (jacobian).inverse * (y_l - y_g)

        x_g_i = (Matrix.column_vector(x_g) + del_x_g)
        x_l_i = (Matrix.column_vector(x_l) - del_x_g)

        # Step-size control
        modfac = 1.0
        und_relax_fac = 0.3

        while ((x_g_i.to_a.flatten.min < 0.0) or
              (x_l_i.to_a.flatten.min < 0.0)) # check if unphysical
            x_g_i = x_g_i - del_x_g # roll back
            x_l_i = x_l_i + del_x_g # roll back

            del_x_g = und_relax_fac*del_x_g # moderate step

            x_g_i = x_g_i + del_x_g # apply moderated step
            x_l_i = x_l_i - del_x_g # apply moderated step
            modfac = modfac*und_relax_fac
        end

        x_g = x_g_i.to_a.flatten
        x_l = x_l_i.to_a.flatten


        b_l = b.eval(all_but_x.merge(n_vec=>x_l[1..-1]))
        b_g = b.eval(all_but_x.merge(n_vec=>x_g[1..-1]))
        safetyrat_l = x_l[0]/b_l # check hard-sphere volume
        safetyrat_g = x_g[0]/b_g

        if safetyrat_l < 1.2 or safetyrat_g < 1.2
        # this iteration has gone wrong
            res_norms << 100.0
            break
        end

        new_newton_func = Matrix.column_vector(y.eval(
```

```
                all_but_x.merge({v=>x_l[0], n_vec=>x_l[1..-1]}))) -
            Matrix.column_vector(y.eval(
                all_but_x.merge({v=>x_g[0], n_vec=>x_g[1..-1]})))

        res_norms << new_newton_func.to_a.flatten.inject(0){|sum, i|
                                                            sum+i.abs}


        # Termination check
        if ii > 1 # Either it has converged, or it has stalled,
        # or it has reached an incredible number of iterations

            if ii == max_iter/2
                puts "iteration taking longer than normal"
            end

            # check if the error tolerance is reached
            if res_norms[-1] < tol
                converged = true
                finished = true
                puts "converged"
            else
                # check if max_iter has been reached
                if ii==max_iter
                    finished = true
                    converged = false
                    puts "max_iter reached"
                else # check if it has stalled
                    if ii > 3
                        average = res_norms[-4..-1].inject(0.0){|sum, e|
                                                                sum+e}/4.0

                        average_change = res_norms[-4..-1].collect{|e|
                                    (e-average).abs}.inject(0.0){|sum, e|
                                                                sum+e}/4.0

                        av_rel_change = average_change/average

                        if av_rel_change < 0.001 # then it has stalled
                            finished = true
                            converged = false
                            puts "iteration stalled"
                        end
                    end
                end
            end
        end

    end

```

```
    if converged
        is2phase = true
        p res_norms
    else
        is2phase = false
    end

    if is2phase # converged
        y_g = (Matrix.column_vector(y.eval(all_but_x.merge({v=>x_g[0],
                        n_vec=>x_g[1..-1]})))).to_a.flatten

        # Always update the guess with the last converged phase calculation
        if guess != nil
            guess[0..-1] = x_l[0..-1]
        end

        out = [t_val, v_val, x_g, x_l, -y_g[0], is2phase]
    else # not converged
        y_g = [-100000.0]
        out = [t_val, v_val, [v_val, *n_vals],
              [0.0, *([0.0]*n_vals.length)], -y_g[0], is2phase]

    end

    if points.respond_to?(:each) and is2phase
        points << out
    end

    return out
    # [temperature, volume, x_gas, x_liq, pressure, is two phase?]

end
```

source/numphases.rb

Appendix D

Gradient calculation verification

verifying_grad.rb

```
#!/usr/bin/env ruby

require './mdo_main.rb'

##############################################
# BUILDING THE MODEL USING EXPRESSION OBJECTS #
##############################################

# CONSTANTS AND VARIABLES
######################
r = MDO::const(8.314)
c = Dim::new # the number of components defined as a dimension

t = MDO::new # temperature
v = MDO::new # volume

n_vec = MDO::new(c) # mole number vector

tc_vec = MDO::new(c) # critical temperature vector
pc_vec = MDO::new(c) # critical pressure vector

# RK-EOS SPECIFIC
######################
b_vec = 0.0867*r*tc_vec/pc_vec # vector of b values for each component
a_vec = 0.4278*r**2.0*tc_vec**2.5/pc_vec # vector of a values for each
    component

k_mat = MDO::new(c, c) # interaction parameter matrix
```

```
28
29
30 a_mat = (a_vec.bc(0, c)*a_vec.bc(c, 0))**0.5*(1.0-k_mat)
31
32 b = (n_vec*b_vec).sum
33 a = (n_vec.bc(0, c)*n_vec.bc(c, 0)*a_mat).sum.sum
34
35 n_tot = n_vec.sum
36
37 p_rk = n_tot*r*t/(v-b) - a/(t**0.5*v*(v+b)) # pressure = f(T,V,n) according
      to RK-EOS
38
39 f_res = n_tot*r*t*(v/(v-b)).ln+a/(b*t**0.5)*(v/(v+b)).ln # residual
     helmholtz free energy
40
41 # IDEAL GAS SPECIFIC
42 #######################
43 # Using non-constant heat capacities
44 a_cp = MDO::new(c)
45 b_cp = MDO::new(c)
46 c_cp = MDO::new(c)
47 d_cp = MDO::new(c)
48 e_cp = MDO::new(c)
49 h0 = MDO::new(c)
50 s0 = MDO::new(c)
51
52 t0 = MDO::const(298.0) # K
53 p0 = MDO::const(1.0e5) # 1 bar in Pa
54
55 #  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
56 # integral of cp(T) dT
57 int1 =  a_cp*(t-t0) +
58         b_cp*(t**2.0-t0**2.0)/2.0 +
59         c_cp*(t**3.0-t0**3.0)/3.0 +
60         d_cp*(t**4.0-t0**4.0)/4.0 +
61         e_cp*(t**5.0-t0**5.0)/5.0
62
63 # integral of cp(T)/T dT
64 int2 =  a_cp*(t/t0).ln +
65         b_cp*(t-t0) +
66         c_cp*(t**2.0-t0**2.0)/2.0 +
67         d_cp*(t**3.0-t0**3.0)/3.0 +
68         e_cp*(t**4.0-t0**4.0)/4.0
69
70
71 h = h0 + int1
72 s = s0 + int2
73 mu0_vec = h - t*s
74 mu_vec = mu0_vec + r*t*(n_vec*r*t/(v*p0)).ln
75
```

```
f_ig = -n_tot*r*t + (mu_vec*n_vec).sum

# TOTAL HELMHOLTZ FREE ENERGY
##########################

f = f_ig + f_res

x = MDO::MDO(v) << n_vec

#############################
# FOR USE IN ITERATION     #
#############################

y = f.grad(x)
dydx = y.grad(x)

##########################
# MODEL EVALUATION DATA #
##########################



# System 2: My arbitrary test system
####################

#               CH4          CH3CH3           CH3CH2CH3
# Critical temperature and pressure
# found from Lange's Handbook of Chemistry:
# http://books.google.no/books?id=ln0eAQAAIAAJ
tc_vals = [ 190.6   ,    305.3   ,   369.552    ] # Kelvin
pc_vals = [ 46.1e5  ,    49.0e5  ,   42.4924e5  ] # Pascal

k_vals = [  [0.0,           0.0,            0.0], # interaction
            [0.0,           0.0,            0.0], # parameter values
            [0.0,           0.0,            0.0]   ] # all set to zero

# cp parameters from 300-1000K
#  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
# Adapted from:
# http://www.me.berkeley.edu/gri\_mech/version30/files30/thermo30.dat
a_cp_vals =  [  42.8161,   35.6789      ,   7.76157  ] # J/(K mol)
b_cp_vals =  [  -0.113661 ,   -4.573982e-2 ,   0.219694  ] # J/(K^2 mol)
c_cp_vals =  [  4.08883e-4,   4.983730e-4  ,   5.07651e-5  ] # J/(K^3 mol)
d_cp_vals =  [  -4.0301535e-7 ,   -5.890189e-7 ,   -1.827209e-7  ] # J/(K^4
     mol)
e_cp_vals =  [  1.38589e-10 ,   2.2338535e-10,   7.91070889e-11  ] # J/(K^5
     mol)

# standard enthalpy and
# entropy found from:
```

```
# http://webbook.nist.gov/chemistry/
h0_vals =  [ -78870.0   ,  -84000    ,  -104700 ] # J/mol 298K (g) 1 bar
s0_vals =  [ 186.25,   229.6   ,   269.91 ] # J/(K mol) 1 bar 298K (g)

t_val = 280.0 # K
v_val = 0.07 # m3
n_vals  = [ 80.0    ,    60.0   ,   40.0       ] # moles

all_but_x = { t=>t_val,
        tc_vec=>tc_vals, pc_vec=>pc_vals,
        a_cp => a_cp_vals,
        b_cp => b_cp_vals,
        c_cp => c_cp_vals,
        d_cp => d_cp_vals,
        e_cp => e_cp_vals,
        h0=>h0_vals, s0=>s0_vals,
        k_mat=>k_vals}



def find_jac(x_g, x_tot, expr, params, v_in, n_in)
    dx = 0.0000001
    jac = (0...x_g.length).collect{|i|
              (0...x_g.length).collect{|j|

                x_g_plus = (0...x_g.length)
                    .collect{|e| if e==i; x_g[e]+dx; else x_g[e]; end}  ;

                x_g_minus = (0...x_g.length)
                    .collect{|e| if e==i; x_g[e]-dx; else x_g[e]; end}  ;

                plus = big_f(x_g_plus, x_tot, expr,
                             params, v_in, n_in)[j] ;

                minus = big_f(x_g_minus, x_tot, expr,
                             params, v_in, n_in)[j] ;

                (plus-minus)/(2.0*dx)
                }
            }
end

def func_eval(x_vals, x, func, p_hash)
    to_eval = p_hash.merge({x[0]=>x_vals[0], x[1]=>x_vals[1..-1]})
    return func.eval(to_eval)
end

def diff_dig(level, length, f, p_hash, x, x_vals, dx, indexes = [])
    if level>0
        return (0...length).collect{|i|
```

```
                    diff_dig(level-1, length, f, p_hash,
                    x, x_vals, dx, indexes.dup << i)}
    else
        denom = 1.0
        indexes.each{|i| denom = denom*dx[i]}

        terms = []
        i = indexes[0]
        x_i_plus = (0...length).collect{|e|
            if e==i; x_vals[e]+dx[e]/2.0; else x_vals[e]; end}

        x_i_minus =  (0...length).collect{|e|
            if e==i; x_vals[e]-dx[e]/2.0; else x_vals[e]; end}

        terms = [[1.0, x_i_plus], [-1.0, x_i_minus]]

        indexes[1..-1].each{ |index|
            old_terms = terms.dup

            terms = []
            old_terms.each{ |term|
                factor = term[0]

                x_i_plus = (0...length).collect{|e|
                    if e==index; term[1][e]+dx[e]/2.0;
                    else term[1][e]; end}

                x_i_minus =  (0...length).collect{|e|
                    if e==index; term[1][e]-dx[e]/2.0;
                    else term[1][e]; end}

                terms << [factor*1.0, x_i_plus]
                terms << [factor*-1.0, x_i_minus]
            }

        }

        terms = terms.collect{|t| t[0]*func_eval(t[1], x, f, p_hash)}

        out = (terms.inject(0.0){|sum, elem| sum + elem}) / denom

        return out
    end

end

def num_ho_grad(x_vars_hash, order, f, p_hash, dx) # look at calls
# below for example usage

    x_vals = x_vars_hash.values.flatten
```

```
    x = x_vars_hash.keys

    if !dx.respond_to?(:each)
        old_dx = dx
        dx = x_vals.collect{old_dx}
    end

    structure = diff_dig(order, x_vals.length, f, p_hash, x, x_vals, dx)

    return structure

end

def ana_ho_grad(x_diff, order, x_vars_hash, f, p_hash)

    argument = x_vars_hash.merge(p_hash)

    diffed = [f.grad(x_diff)]
    (1...(order)).each{|i| diffed[i] = diffed[i-1].grad(x_diff)}

    return diffed[-1].eval(argument)
end


dv = 1e-3
dn = 1e-0

first_num  =  num_ho_grad({v=>v_val, n_vec=>n_vals}, 1, f,
                         all_but_x, [dv, dn, dn, dn])

second_num =  num_ho_grad({v=>v_val, n_vec=>n_vals}, 2, f,
                         all_but_x, [dv, dn, dn, dn])

third_num  =  num_ho_grad({v=>v_val, n_vec=>n_vals}, 3, f,
                         all_but_x, [dv, dn, dn, dn])

#fourth_num =  num_ho_grad({v=>v_val, n_vec=>n_vals}, 4, f,
#                          all_but_x, [dv, dn, dn, dn])

first_ana  = ana_ho_grad(x, 1, {v=>v_val, n_vec=>n_vals}, f, all_but_x)
second_ana = ana_ho_grad(x, 2, {v=>v_val, n_vec=>n_vals}, f, all_but_x)
third_ana  = ana_ho_grad(x, 3, {v=>v_val, n_vec=>n_vals}, f, all_but_x)
#fourth_ana = ana_ho_grad(x, 4, {v=>v_val, n_vec=>n_vals}, f, all_but_x)

grad1_comp = first_ana.zip(first_num).collect{|a, b| a/b}
p grad1_comp
puts "max deviation = #{grad1_comp.collect{|e| (e-1.0).abs}.max}"
puts ""

grad2_comp = (0...4).collect{|i1| (0...4).collect{|i2| second_ana[i1][i2]/
```

```
    second_num[i1][i2]}}
p grad2_comp
puts "max deviation = #{grad2_comp.flatten.collect{|e| (e-1.0).abs}.max}"
puts ""

grad3_comp = (0...4).collect{|i1| (0...4).collect{|i2| (0...4).collect{|i3|
     third_ana[i1][i2][i3]/third_num[i1][i2][i3]}}}
p grad3_comp
puts "max deviation = #{grad3_comp.flatten.collect{|e| (e-1.0).abs}.max}"
puts ""

#grad4_comp = (0...4).collect{|i1| (0...4).collect{|i2| (0...4).collect{|i3
    | (0...4).collect{|i4| fourth_ana[i1][i2][i3][i4]/fourth_num[i1][i2][i3
    ][i4]}}}}
#p grad4_comp
#puts "max deviation = #{grad4_comp.flatten.collect{|e| (e-1.0).abs}.max}"
#puts ""
```

source/verifying_grad.rb

Appendix E

Calculating points on the phase diagram

phaseTraceBeam.rb

```
#!/usr/bin/env ruby

require './mdo_main.rb'

##############################################
# BUILDING THE MODEL USING EXPRESSION OBJECTS #
##############################################

# CONSTANTS AND VARIABLES
######################
r = MDO::const(8.314)
c = Dim::new # the number of components defined as a dimension

t = MDO::new # temperature
v = MDO::new # volume

n_vec = MDO::new(c) # mole number vector

tc_vec = MDO::new(c) # critical temperature vector
pc_vec = MDO::new(c) # critical pressure vector

# RK-EOS SPECIFIC
######################
b_vec = 0.0867*r*tc_vec/pc_vec # vector of b values for each component
a_vec = 0.4278*r**2.0*tc_vec**2.5/pc_vec # vector of a values for each
    component

k_mat = MDO::new(c, c) # interaction parameter matrix
```

```
28
29
30 a_mat = (a_vec.bc(0, c)*a_vec.bc(c, 0))**0.5*(1.0-k_mat)
31
32 b = (n_vec*b_vec).sum
33 a = (n_vec.bc(0, c)*n_vec.bc(c, 0)*a_mat).sum.sum
34
35 n_tot = n_vec.sum
36
37 p_rk = n_tot*r*t/(v-b) - a/(t**0.5*v*(v+b)) # pressure = f(T,V,n) according
      to RK-EOS
38
39 f_res = n_tot*r*t*(v/(v-b)).ln+a/(b*t**0.5)*(v/(v+b)).ln # residual
     helmholtz free energy
40
41 # IDEAL GAS SPECIFIC
42 #######################
43 # Using non-constant heat capacities
44 a_cp = MDO::new(c)
45 b_cp = MDO::new(c)
46 c_cp = MDO::new(c)
47 d_cp = MDO::new(c)
48 e_cp = MDO::new(c)
49 h0 = MDO::new(c)
50 s0 = MDO::new(c)
51
52 t0 = MDO::const(298.0) # K
53 p0 = MDO::const(1.0e5) # 1 bar in Pa
54
55 #  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
56 # integral of cp(T) dT
57 int1 =  a_cp*(t-t0) +
58         b_cp*(t**2.0-t0**2.0)/2.0 +
59         c_cp*(t**3.0-t0**3.0)/3.0 +
60         d_cp*(t**4.0-t0**4.0)/4.0 +
61         e_cp*(t**5.0-t0**5.0)/5.0
62
63 # integral of cp(T)/T dT
64 int2 =  a_cp*(t/t0).ln +
65         b_cp*(t-t0) +
66         c_cp*(t**2.0-t0**2.0)/2.0 +
67         d_cp*(t**3.0-t0**3.0)/3.0 +
68         e_cp*(t**4.0-t0**4.0)/4.0
69
70
71 h = h0 + int1
72 s = s0 + int2
73 mu0_vec = h - t*s
74 mu_vec = mu0_vec + r*t*(n_vec*r*t/(v*p0)).ln
75
```

```
f_ig = -n_tot*r*t + (mu_vec*n_vec).sum

# TOTAL HELMHOLTZ FREE ENERGY
##########################

f = f_ig + f_res

x = MDO::MDO(v) << n_vec

#############################
# FOR USE IN ITERATION     #
#############################

y = f.grad(x)
dydx = y.grad(x)

##########################
# MODEL EVALUATION DATA #
##########################



# System 2: My arbitrary test system
####################

#               CH4          CH3CH3           CH3CH2CH3
# Critical temperature and pressure
# found from Lange's Handbook of Chemistry:
# http://books.google.no/books?id=ln0eAQAAIAAJ
tc_vals = [ 190.6   ,    305.3   ,   369.552    ] # Kelvin
pc_vals = [ 46.1e5  ,    49.0e5  ,   42.4924e5  ] # Pascal

k_vals = [  [0.0,           0.0,            0.0], # interaction
            [0.0,           0.0,            0.0], # parameter values
            [0.0,           0.0,            0.0]   ] # all set to zero

# cp parameters from 300-1000K
#  Cp  = A + B*t + C*t^2 + D*t^3 + E*t^4
# Adapted from:
# http://www.me.berkeley.edu/gri\_mech/version30/files30/thermo30.dat
a_cp_vals =  [  42.8161,   35.6789      ,   7.76157  ] # J/(K mol)
b_cp_vals =  [  -0.113661 ,   -4.573982e-2 ,   0.219694  ] # J/(K^2 mol)
c_cp_vals =  [  4.08883e-4,   4.983730e-4  ,   5.07651e-5  ] # J/(K^3 mol)
d_cp_vals =  [  -4.0301535e-7 ,   -5.890189e-7 ,   -1.827209e-7  ] # J/(K^4
     mol)
e_cp_vals =  [  1.38589e-10 ,   2.2338535e-10,   7.91070889e-11  ] # J/(K^5
     mol)

# standard enthalpy and
# entropy found from:
```

```
# http://webbook.nist.gov/chemistry/
h0_vals =  [ -78870.0   ,  -84000    ,  -104700 ] # J/mol 298K (g) 1 bar
s0_vals =  [ 186.25,   229.6   ,   269.91 ] # J/(K mol) 1 bar 298K (g)

#           CH4          CH3CH3           CH3CH2CH3
n_vals = [ 80.0    ,    60.0   ,   40.0       ] # mixture of hydrocarbons

all_but_x = {tc_vec=>tc_vals, pc_vec=>pc_vals,
         a_cp => a_cp_vals,
         b_cp => b_cp_vals,
         c_cp => c_cp_vals,
         d_cp => d_cp_vals,
         e_cp => e_cp_vals,
         h0=>h0_vals, s0=>s0_vals,
         k_mat=>k_vals}

require './numphases.rb'

# before starting to call numphases, set up a reference to a guess variable
guess = [] # empty array, to be filled by the
# numphases calculation when it converges

points = [] # empty array, to be filled by the
# numphases calculation when it converges

# First find upper or lower phase boundary
# then increase temperature and repeat

# starting point chosen freely
start = [280.0, 0.05, true]

start_phase = numphases(t, start[0], v, start[1], n_vec, n_vals, all_but_x,
      b, y, dydx, guess, points)

# confirm that we started inside the phase envelope
start[-1] = start_phase[-1]

# the direction to move after the phase boundaries
# have been found for this temperature
dir = [1.0, 0.0]

# will denote the upper and lower volume checked
vol_pairs = []


starts = []
starts << start_phase[3].dup

while true
    step_vol = 0.1
```

```
    # find a point above that doesn't converge
    top = [start[0], start[1], true]

    while top[-1]==true
        top[1] = top[1]+step_vol
        top_phase = numphases(t, top[0], v, top[1], n_vec, n_vals,
                              all_but_x, b, y, dydx, guess, points)
        top[-1] = top_phase[-1]
    end

    # the bot point is known to be outside the two phase region
    bot = [start[0], 0.008, false]

#    puts "here"
#    p start
#    p top
#    p bot

    ind = -1
    bot_top_vols = []
    [bot, top].each{|other_point|
        ind = ind+1
        pinch = [start.dup, other_point.dup]
        del_vol = pinch[1][1]-pinch[0][1]

        # reset guess
        guess = []

        tol1 = 0.003 # m3

        p pinch

        # "divide and conquer phase"
        while del_vol.abs > tol1
            mid = []
            mid[0..-1] = pinch[0][0..-1]
            mid[1] = mid[1]+del_vol/2.0

            mid[-1] = numphases(t, mid[0], v, mid[1], n_vec, n_vals,
                                all_but_x, b, y, dydx, guess, points) [-1]

            if mid[-1] == pinch[0][-1]
                pinch[0][0..-1] = mid[0..-1]
            else
                pinch[1][0..-1] = mid[0..-1]
            end

            del_vol = pinch[1][1]-pinch[0][1]
            p pinch
```

```
        end


        # dump points to a file
        if !FileTest.exist?("beamresult.m")
            file = File.open("beamresult.m", "w")
            file.close
        end
        dump = File.open("beamresult.m", "w")
            dump.puts "resultvec = ["+points.collect{|e|
                e.flatten[0..-2].join(" ") }.join("; ")+"]; "
        dump.close


        # "shooting phase"
        # take the first point, go up a tiny bit,
        # if it is outside the two-phase region, roll back
        # use half step length
        dv = del_vol
        next_point = [pinch[0][0], pinch[0][1], false]

        tol2 = 0.00001

        while dv.abs > tol2
            next_point[1] = next_point[1]+dv
            next_point[-1] = numphases(t, next_point[0], v, next_point[1],
                                    n_vec, n_vals, all_but_x, b, y, dydx,
                                    guess, points) [-1]

            if next_point[-1] == false
                # roll back and divide dv by 2.0
                next_point[1] = next_point[1]-dv

                dv = dv/2.0

                puts "overshot"
            end

            p next_point
        end

        bot_top_vols[ind]= next_point[1]

        dump = File.open("beamresult.m", "w")
            dump.puts "resultvec = ["+points.collect{|e|
                e.flatten[0..-2].join(" ") }.join("; ")+"]; "
        dump.close
    }
    # Now both the top and bottom boundary of the
    # two phase region has been found and saved in
```

```
    # "bot_top_vols"

    # update dir
    if vol_pairs.length > 0
        # go in the direction that the phase boundaries move
        old_av = (vol_pairs[-1][1]+vol_pairs[-1][0])/2.0
        av_change = (bot_top_vols[1]+bot_top_vols[0])/2.0 - old_av
        dir[1] = av_change

        start = [start[0]+dir[0], old_av+dir[1], true]

    else
        start = [start[0]+dir[0], start[1]+dir[1], true]
    end

    # confirm that the next starting point (=the last +dt)
    # is inside the two phase region
    guess = starts[-1]

    start_phase = numphases(t, start[0], v, start[1], n_vec, n_vals,
                            all_but_x, b, y, dydx, guess, points)

    start[-1] = start_phase[-1]

    while start[-1]==false # if it is not inside
        dir[0] = dir[0]/2.0 # halve the distance moved
        dir[1] = dir[0]/2.0

        start = [start[0]-dir[0], start[1]-dir[1], true]
        guess = starts[-1]
        start_phase = numphases(t, start[0], v, start[1], n_vec, n_vals,
                                all_but_x, b, y, dydx, guess, points)
    end

    starts << start_phase[3].dup

    vol_pairs << bot_top_vols
end
```

source/phaseTraceBeam.rb