Pergamon

PII: S0098-3004(96)00030-1

Computers & Geosciences Vol. 22, No. 7, pp. 801–815, 1996

0098-3004/96 $15.00 + 0.00

A NEW OBJECT-ORIENTED LIBRARY FOR CALCULATING ANALYTICALLY HIGH-ORDER MULTIVARIABLE DERIVATIVES AND THERMODYNAMIC PROPERTIES OF FLUIDS WITH EQUATIONS OF STATE

RÉGIS THIÉRY
CREGU, BP 23, F-54501 Vandoeuvre-lès-Nancy Cedex, France

(*e-mail*: thiery@cregu.cnrs-nancy.fr)

(Received 25 May 1995; revised 13 January 1996)

Abstract—A new global approach for calculating thermodynamic properties of fluids with equations of state is proposed. This method uses an object-oriented library, written in C++, and containing a number of routines that facilitate thermodynamic calculations. It is applicable to any equation of state formulated by an analytical expression of the Helmholtz free energy. The programming work of the user is limited to the building of a parse representation of the Helmholtz free energy. This parse graph is then processed by the program for analytically calculating the required derivatives and thermodynamic properties of fluids. A demonstration of this library is made with the Anderko and Pitzer equation of state for H_2O–NaCl–KCl fluids. The Helmholtz free energy has been differentiated analytically up to the fourth order with respect to the temperature, molar volume, and composition. Calculated derivatives are used to calculate the critical line of H_2O–NaCl mixtures and other thermodynamic properties, which otherwise would be difficult to obtain. Copyright © 1996 Elsevier Science Ltd

Key Words: Equations of state, Thermodynamics, Critical state, H_2O–NaCl fluids, Object-oriented programming.

INTRODUCTION

Knowledge of the thermophysical properties of geological fluids is required for the understanding of chemical and mechanical processes occurring in various environments. The equilibrium state of a fluid mixture of N components is defined entirely by a set of $(N+1)$ intensive variables that can be either TVx_i, PTx_i, SVx_i, or SPx_i (where T is the temperature, V is the molar volume, P is the pressure, S is the molar entropy of the system, and x_i is the mole fraction of the ith component in the mixture). Any expression of the form $F(a_1,a_2,...) = 0$ in which the a_i are the $(N+1)$ variables defining the equilibrium state of the system is termed an equation of state (EOS). Its formulation can be empirical or guided by theoretical expressions. In TVx_i coordinates, EOS may be given by expressions of the Helmholtz free energy, A from which all desired quantities can be obtained by classical thermodynamic relationships.

$$A = A(T,V,x_i). \tag{1}$$

Such a formulation, referred to as the fundamental equation, is more convenient than $P = P(T,V,x_i)$ as all thermodynamic properties of the fluid are obtained by simple differentiation with respect to the TVx_i variables. This formulation is now used by most recently published EOS, and has been chosen for this study.

A great deal of the work of the thermodynamist consists of differentiating the fundamental equation to obtain all the desired thermodynamic properties. This can represent a large amount of time, as specific mathematical formulations of EOS are rarely simple. Recent advances in statistical mechanics lead to efficient, but complex expressions, which are cumbersome to differentiate. A good illustration of this is given by the Anderko and Pitzer EOS (1993a, 1993b) for the H_2O–NaCl–KCl system above 300°C. This EOS is successful in calculating liquid–vapor phase equilibria and adequately predicts the densities of liquids and vapors. However, for deriving the molar enthalpy, H, from the Helmholtz free energy (Eq. 2), Anderko and Pitzer remark that "the temperature dependency of some of the terms of A is so complex that the derivative was determined numerically rather than analytically,"

$$H = -T^2\left(\frac{\partial(A/T)}{\partial T}\right)_{V,x_i} - V\left(\frac{\partial A}{\partial V}\right)_{T,x_i} \tag{2}$$

Generally, other important parameters such as the pressure and the chemical potential of a component are easy to obtain by simple first-order differentiation:

$$P = -\left(\frac{\partial A}{\partial V}\right)_{T,x_i} \tag{3a}$$

$$\mu_i = \left(\frac{\partial A}{\partial n_i}\right)_{T,V,n_j} = \frac{1}{n}\left[\left(\frac{\partial A}{\partial x_i}\right)_{T,V,x_j} - \sum_k \left(\frac{\partial A}{\partial x_k}\right)_{T,V,x_j}\right], \quad (3b)$$

where n and n_i stand, respectively, for the total number of moles and the number of moles of the ith component in the system. The complexity of expressions increases rapidly with the order of differentiation (see later), and differentiation up to the fourth order is required. Indeed, efficient algorithms for calculating fluid phase equilibria (Peng and Robinson, 1977; Asselineau, Bogdanic, and Vidal, 1979) require second-order differentiates of the Helmholtz free energy. Critical phenomena play an important role in geochemical processes (Bischoff, Rosenbauer, and Pitzer, 1986; Johnson and Norton, 1991), and the location of critical points of fluid mixtures requires differentiates at least up to the third order (Huron, Dufour, and Vidal, 1977; Baker and Luks, 1978; Heidemann and Khalil, 1980; Michelsen, 1980; Michelsen and Heidemann, 1981; Eaton, 1988).

There are three methods to obtain analytical expressions of the derivatives of the free energy. The first one is hand calculation, which can be lengthy and painstaking. The second one is numerical differentiation. As shown previously, this method is sometimes used for first-order differentiate calculations, but should be discarded for calculating higher-order differentiates. Three-point and five-point finite-centered formulae are not reliable enough for determining the second- and third-order differentiates, which are necessary for locating a critical point (Heilig, 1988; Heilig and Franck, 1990). Differentiates from the third order up to the sixth order have been obtained by Deiters and Pegg (1989) by using optimized numerical differentiation formulas, but this requires more computing time. The third method involves the use of computer programs that perform analytical calculations, such as Mathematica (Wolfram, 1991), Reduce (Hearn, 1987), Matlab (Chen and Moler, 1993). Such programs are termed symbolic calculation software, and are used with some success (Heilig, 1988; Heilig and Franck, 1990; Deiters, pers. comm., 1994); however, they are not free of critics. The main problem is that these programs require extensive of memory for complicated EOS. Thus, the analytical expressions of derivatives that can be produced with these programs are limited mostly to the third order (Deiters, pers. comm., 1994). Therefore, the creation of new programming tools devoted specifically to EOS would be a great help.

This paper describes a new tool, which is now used in our laboratory for calculating analytically the differentiates of the Helmholtz free energy and the thermodynamic properties of fluids. This method uses an object-oriented library, written in C++. It is based on an algorithm which looks similar to the one used by symbolic calculation software. However, our library overcomes the shortcomings of these programs and is able to handle the most complex EOS and to calculate multivariable differentiates up to the fourth order (and higher, if required) with respect to the T, V and x_i. First, a brief explanation of the differentiation algorithm will be given. Differences between our library and symbolic calculation programs are pointed out. Then, the use of this library is illustrated with the Anderko and Pitzer EOS (1993a, b).

PRINCIPLES OF THE DIFFERENTIATION ALGORITHM

The difficulty of differentiating a complex arithmetic expression is the number of terms, which grows exponentially each time the order of differentiation is incremented (Norris, 1981). For a small subexpression such as the product of two multivariable functions, such as $u*v$, the first-order derivative involves 4 terms, the second order 8 terms, the third order 16 terms, etc. and hence, the number of operations is multiplied by two at each differentiation step. The situation is worse for the division operator, where the number of operations is multiplied practically by three at each step. Thus, even after simplification and factorization of the expressions, the fourth-order derivative of u/v involves at least 131 terms. Most specific statements of the Helmholtz free energy function contain many terms and operators. Some of these statements have been listed in Table 1 together with the number of mathematical operations required for calculating the Helmholtz free energy. Cubic EOS require relatively few operations, which make them useful for practical calculations (Thiéry,

Table 1. Number of mathematical operations required for calculating Helmholtz free energy of some EOS

EOS	No.
Cubic EOS, attractive part	10
Boublik EOS (1970) for hard-spheres mixtures	20
Anderko and Pitzer, 1993a), dipole–dipole interaction for H_2O–NaCl system	21
Anderko and Pitzer, 1993a), dispersion interactions for H_2O–NaCl system	52
IUPAC EOS for CH_4(Angus, Armstrong, and de Reuck, 1978)	53
IUPAC EOS for CO_2(Angus, Armstrong, and de Reuck, 1976)	58
Keenan EOS (1969) for H_2O	55
Haar–Gallager–Kell EOS (1984) for H_2O	85
Hill EOS (1990) for H_2O (outside the critical region)	91
The MSA ion-dipole model (Lvov and Wood, 1990) for H_2O–NaCl solutions	133

Vidal, and Dubessy, 1994; Thiéry, Kerkhof, and Dubessy, 1994; Thiéry and Dubessy, 1996). However, some recently published EOS (Angus, Armstrong, and de Reuck, 1976, 1978; Keenan and others, 1969; Boublik, 1970; Haar, Gallagher, and Kell, 1984; Hill, 1990; Anderko and Pitzer, 1993a, b) require many more operations. Now, new efficient EOS (Lvov and Wood, 1990; Harvey, 1991), which are based on recent advances of statistical mechanics, are more complex still. Thus, it is not a surprise that the differentiation of such a function becomes quickly impracticable.

A first answer to this type of problem has been given by symbolic calculation software. These programs are all based on an algorithm used widely in computer science (Haugeland, 1985; Sedgewick, 1992). The principle of this algorithm is simple: a complex task can always be decomposed into elementary operations and decomposition is conveniently visualized by a hierarchical representation. For instance, consider how a simple arithmetic expression is split into smaller subexpressions (Fig. 1A):

$$f(x,y,z) = u \times v + v/u \times w\,, \qquad (4)$$

where f, u, v and w are a function of three variables, x, y and z. This is how symbolic calculation programs store an arithmetic expression in memory. This type of representation is termed a parse tree (Sedgewick, 1992). Each box is termed a node and refers to an arithmetic operation (an addition, a multiplication, the logarithm of a quantity, etc.). The upper node is the root and it contains the expression to be decomposed. Bifurcation starts from the root and continues so long as it is possible to decompose a subexpression further. Terminal nodes are termed the leaves and contain irreducible expressions. Now, for calculating a first-order differentiate, symbolic calculation programs traverse the parse tree from the leaves up to the root. At each node of the parse tree, a simple first-order derivation rule is applied. The result is the construction of a new parse tree. Figure 1B shows the parse tree obtained for the first-order partial derivative $(\partial f/\partial x)_{y,z}$ denoted as f_x (derivatives are written using the convention, where subscripts designate the differentiation variables). The root of this new parse tree contains the desired arithmetic expression of the first-order differentiate. It is worth noting that the parse tree of a first-order differentiated expression already grows considerably in size. For calculating a derivative of order n, symbolic calculation programs apply this algorithm n times. Thus, there is no theoretical limit for the order of differentiates that can be calculated. However, memory requirements of this algorithm may be so great that expressions of fourth- and higher-order differentiates cannot be obtained (Deiters, pers. comm., 1994).

This leads us to think about an alternative method for calculating analytical values of differentiates. Our method uses parse representations, too; but there are large differences with respect to the method used by symbolic calculation software. As shown above, these programs manipulate only strings of characters, which are arithmetic expressions of the derivatives. These strings of characters then can be inserted in the source code of the calculation programs. On the contrary, our library does not try to obtain the analytical expressions of the partial derivatives, it calculates only analytically their numerical values. Symbolic calculation software is able only to apply first-order derivation rules to a given parsed expression. In our library package, the source code for calculating the first-, second-, third-, and fourth-order derivatives has been written for a large number of operators. Such a mode of operation has several advantages. First, memory requirements of our library are smaller. For each derivative, symbolic calculation software produce a new parse tree, which is progressively more complex each time that the differentiation order is increased (Fig. 1B). With this library, no parse tree for the partial differentiates is constructed. Indeed, for calculating a differentiate of order n, the library uses only the parse tree of the arithmetic expression, that has been built by the user (Fig. 1C). Of course, the order of differentiates that can be obtained at this time is limited to the fourth order, but, if necessary, this limit could be increased to a higher order. It would require to write only the source code of higher derivatives. Another shortcoming of symbolic calculation programs is that they give complex and lengthy analytical expressions for the differentiated expressions. Many repetitions can be observed in the source code produced by symbolic calculation software. These expressions should be simplified and factorized in order to produce an efficient and rapid program, but, as pointed out by specialists of symbolic calculation, the automatic simplification of arithmetic expressions remains a difficult problem of computer science. For now, symbolic calculation software is not able to factorize simple expressions effectively. In our library, this problem is solved, as simplifications and factorizations have been achieved at the stage of writing the source code.

Our library has been written in C++ (Stroustrup, 1991), an object-oriented extension of the C language (Kernighan and Ritchie, 1978). Because users of this library will have to write some instructions in C++, it is important to explain how this library is organized and constructed.

ORGANIZATION OF THE LIBRARY

This library (Fig. 2) is composed of two modules interacting with each other: the first module applies to the calculation of analytical differentiates, the second module calculates thermodynamic properties of fluids. The first module contains only source code for calculating the differentiates of a generic product,

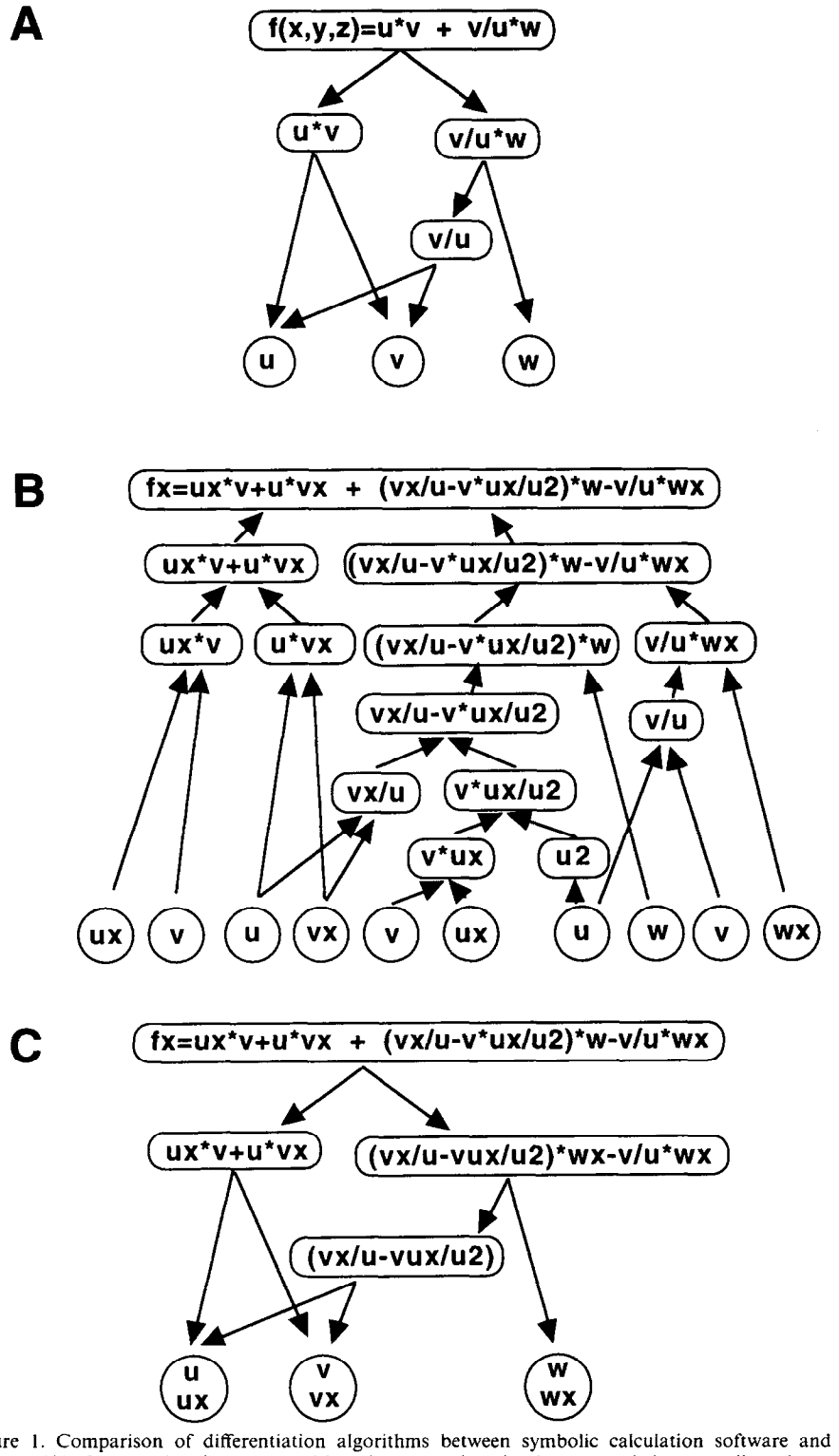

Figure 1. Comparison of differentiation algorithms between symbolic calculation software and our library. (A) Start point is same: arithmetic expression is decomposed into smaller elementary subexpressions (boxes) which are easier to process. Irreducible subexpressions, called "leaves", are represented by circles. Arrows and numbers indicate direction and order of the decomposition (from the root up to leaves). (B) Symbolic calculation software builds new parse tree for each differentiate. Here is represented parse tree of differentiated function with respect to one variable. Parse tree is obtained by applying simple first-order differentiation rules on parse tree of (A). Growth of parse tree starts from leaves and ends at root. (C) Our library uses same parse tree as (A) for calculating any differentiate of n-order. Differentiates are obtained by applying differentiation rules that have been programmed in library. Notations ux, vx, wx, u2 are used, respectively, for u_x, v_x, w_x and u^2 functions.

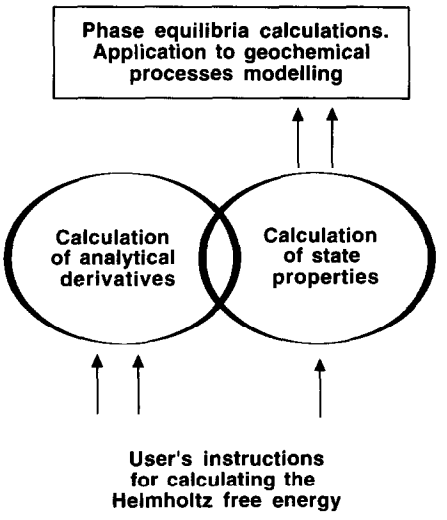

Figure 2. Organization of library. Two main modules make up core of library: (1) calculation of differentiates of simple expressions, like product of two functions etc. and (2) calculation of state properties of fluids as a function of partial derivatives of Helmholtz free energy. Library communicates by means of (1) user's instructions for constructing parse tree of any EOS, and with (2) other specialized libraries of geochemical calculations.

a sum, a ratio etc. In the second module, all state parameters of interest for the thermodynamics of fluid have been expressed as a function of partial differentiates of the Helmholtz free energy. The source code is applicable to any EOS, expressed by means of the Helmholtz free energy. The first step for using this library is to write C++ instructions for constructing the parse tree of the specific Helmholtz free energy function of interest. Then, the user must compile the source file with a C++ compiler and link it with the library. After this step, the user can call all the available subroutines of thermodynamic calculations. This library and the user's code can be linked to other specialized libraries for geochemical process modelling.

USE OF THE DIFFERENTIATION LIBRARY

A detailed example

The use of the differentiation module will be illustrated with the simple function, given by Equation (4). Using C++, one defines and manipulates entities, termed objects, that combine variables and functions. Thus, the first step is the creation of an "object". This is what is shown in the following example. The keyword "`struct`" indicates that a new object is being defined. "`AnExample`" is the name of the new defined object. Variables and functions are declared between the brackets. Comments are introduced with the characters "//".

```
//-------------------------------------------------
      struct AnExample : public Equation
//-------------------------------------------------
      {
      // Declaration of the nodes of the parse tree

      Sum2 f ;                  // f = uv + vuw
      Product uv ;              // uv = u * v
      Product vuw ;             // vuw = vu * w
      Division vu ;             // vu = v/u

      // Declaration of the leaves of the parse tree
      //    u(x,y,z) = 2.0 * x * x ;
      //    v(x,y,z) = x + y + z
      //    w(x,y,z) = x - y * z

      Leaf u ;
      Leaf v ;
      Leaf w ;

      // The differentiation variables

      double x ;
      double y ;
      double z ;

      // Declaration of the functions

      AnExample () ;
      void calculate_the_leaves () ;
      void calculate_the_partial_differentiates () ;
      void check_derivatives () ;
      } ;
```

The second step is to construct a C++ representation of the parse tree. The method used here is similar to that used by symbolic calculation softwares. This is performed in our example by the function `AnExample ()`.

```
//-------------------------------------------------
      AnExample :: AnExample () : Equation ()
//-------------------------------------------------
//    Construction of the parse tree (Fig 1A)
//-------------------------------------------------
{
f . tie ( uv , vuw ) ;                    // f = uv + vuw
f . set_scalars ( 1.0 , 1.0 , 0.0 ) ;

uv  . tie ( u , v ) ;                     // uv = u * v
vuw . tie ( vu , w ) ;                    // vuw = vu * w

vu . tie ( v , u ) ;                      // vu = v / u

//    There are three differentiation variables (x,y and z)

f . number_of_variables (3) ;
}
```

The `tie ()` function connects the nodes. After the parse graph has been defined entirely, the number of differentiation variables must be specified by calling the `number_of_variables ()` function.

Finally, in the third step, the user writes the function that calculates the partial differentiates of irreducible expressions. This is carried out by the `calculate_the_leaves ()` function.

Variables for which derivatives are to be calculated are given arbitrary identification numbers, starting from zero. These numbers are used as index numbers for referencing a specific derivative value. An important remark is given here: the user must specify differentiation variables of any derivative in a decreasing order (for example, for referring to the w_{xy} derivative, and by using the following identification numbers (x,0), (y,1) and (z,2), the user will have to write `w(2,1)` and not `w(1,2)`).

```
//-------------------------------------------------------------
      void AnExample :: calculate_the_leaves ()
//-------------------------------------------------------------
//    Calculation of partial differentiates of irreductible
//    expressions
//    Indexes of the differentiation variables are :
//          0 : the x variable
//          1 : the y variable
//          2 : the z variable
//-------------------------------------------------------------
{
//    u(x,y,z) = 2.0 * x * x ;
//    v(x,y,z) = x + y + z
//    w(x,y,z) = x - y * z


u () = 2.0 * x * x ;
v () = x + y + z ;
w () = x - y * z ;

u (0) = 2.0 * x ;
u (0,0) = 2.0 ;

v (0) = 1.0 ;
v (1) = 1.0 ;
v (2) = 1.0 ;

w (0) = 1.0 ;
w (1) = -z ;
w (2) = -y ;
w (2,1) = -1.0 ;

// Other derivatives are equal to zero, by default.
}
```

Now, it remains to call the library functions that calculate the partial differentiates. This is performed by the following three instructions in the `calculate_the_partial_differentiates ()` function.

```
//-------------------------------------------------------------
      void AnExample :: calculate_the_partial_differentiates ()
//-------------------------------------------------------------
//    Calculation of the partial differentiates.
//-------------------------------------------------------------
{
int order = 4 ;   // the differentiation order

f . initialize_the_graph () ;

calculate_the_leaves () ;

f . calculate ( order ) ;
}
```

The `initialize_the_graph ()` function resets an internal flag that prevents repetitive expressions from being recalculated.

Here is the subroutine for calculating and displaying the f_{xy} and f_{xyz} partial differentiates:

```
//-------------------
      void main ()
//-------------------
{
AnExample example ;

example . calculate_the_partial_differentiates () ;

printf ("%Lg %Lg" , example . f (1,0) , example . f (2,1,0) ) ;
}
```

Some useful functions

In this section, some useful functions available within the differentiation library are addressed. First, it may happen that some irreducible expressions do not depend on some variables. For instance, in the previous model, where the leaf referred to as u does not depend on the y and z variables. Thus, calculation of differentiates with respect to these variables is redundant. For this reason, the user has the option to specify to the library whether an expression really is a function of a given variable or not. This is achieved with the following instructions:

```
u . specify_variable_dependency ( 1 , FALSE ) ;
v . specify_variable_dependency ( 2 , FALSE ) ;
      // 1 and 2 are the indexes of the y and z variables
      // in the array of differentiation variables
```

In this example, the user indicates that the expression u is not dependent on the y variable. After all specifications have been set, the user can request an analysis of the dependencies of the nodes of the parse tree by calling the `analyze_dependencies ()` function

```
f . analyze_dependencies ()
```

Another important group of functions furnished by this library concerns the detection of mistakes made by the user. Possible errors are of two types. The first type occurs when the user forgets to define one connection between two nodes. The user should therefore be careful when the parse graph contains many nodes. Two utility functions are given for detecting this type of error (the `print ()` and `check_pointers ()` function).

```
f . print ( file ) ;
f . check_connections ( file ) ;
```

The `print ()` function writes on a file the list of the nodes with their connections. Memory addresses and the nature of the nodes are indicated. The `check_connections ()` function detects whether a connection is lacking. The second type of error involves differentiation mistakes of the leaves. This library provides a method for detecting them. The user can carry out a systematic checking of differentiates of a given node by calling the `check1 ()`, `check2 ()`, `check3 ()`, and `check4 ()` functions, respectively, for checking first-, second-, third, and fourth- order derivatives. Each of these functions compares values of derivatives, calculated with the library and by a finite-centered difference method. Analytical and numerical values are then reported on an ASCII file specified by the user. If the number digits are not identical (except for the last digit), there is an error in the derivatives of the "leaves" expressions.

The list of operators and functions

The list of available operators and functions is given in Table 2, and covers most of the situations encountered in thermodynamic modelling. Note that although exponentiation `Powo` and `Power` objects look similar, they do not have the same behavior. `Power` objects accept only positive values of $(u + b)$ expressions, whereas `Powo` objects accept positive

```
//-------------------------------------------------------
      void AnExample :: check_derivatives ()
//-------------------------------------------------------
{
FILE * report = fopen ( "Derivatives.note" , "w") ;

double var [3], dvar [3] ;

var [0] = 200.0 ;              // the x variable
var [1] = 40.0  ;              // the y variable
var [2] = 500.0 ;              // the z variable

dvar [0] = 1.0e-6 ;     // step parameters used
dvar [1] = 1.0e-6 ;     // for calculating numerical
dvar [2] = 1.0e-6 ;     // differentiates

f . check1 ( report , var , dvar ) ;
f . check2 ( report , var , dvar ) ;
f . check3 ( report , var , dvar ) ;
f . check4 ( report , var , dvar ) ;

fclose (report) ;
}
```

and negative values. However, `Power` objects accept any value for the exponent, and `Powo` objects accept only exponents which can be expressed by a ratio of two integers ($c = n/m$ where c is the exponent, n denotes any integer, and m is an odd integer). For constructing a parse tree for any arithmetic expression, the knowledge of Table 2 and of two important functions (`tie ()` and `set_scalars ()` functions) is sufficient. The `tie ()` function connects a parental node (denoted as `P`) to one or two subexpressions (denoted as `u` and `v`).

```
Node P, u, v ;

P . tie ( u ) ;          // if P connects to only one node

P . tie ( u , v ) ;      // or if P connects to two nodes
```

Arithmetic expressions always involve scalar parameters. These are always specified by calling the `set_scalars ()` function.

```
Node P ;
double a, b, c, d ;     // scalar parameters, as defined
                        // by Table 2

P . set_scalars ( a ) ;
      // if P is a Cube object, for instance

P . set_scalars ( a, b ) ;
      // if P is an Inverse object, for instance

P . set_scalars ( a , b , c ) ;
      // if P is a Power object, for instance

P . set_scalars ( a , b , c , d ) ;
      // if P is an Exponential object, for instance
```

Other operators are more complex, and their use must be defined explicitly. These are the `Sum`, `LineaX2`, `LineaX3`, `SumX2`, `SumX3`, `SumX4`, and `SumX5` types of nodes. They are used mostly to extend EOS to fluid mixtures. They involve multiple summations through the components of the system. The different steps required for using these objects are summarized in Appendix A for the `Sum` object, in Appendix B for the `LineaX` and `LineaX2` objects, and in Appendix C for the `SumX2`, `SumX3`, `SumX4` and `SumX5` objects.

USE OF THE MODULE FOR THERMODYNAMIC CALCULATIONS

In representing and calculating the thermodynamic properties of fluids, it is usual to split molecular interactions within a fluid into several contributions; for instance, the repulsive contribution, the dispersion, the dipole–dipole interaction, the dipole–quadrupole contribution,and others. This has been considered in the construction of our library. Two basic objects have been implemented: the `Interaction` and `FluidPhase objects`. The `Interaction` object is used for modelling different types of interactions, whereas the `FluidPhase` object contains all basic variables and functions for implementing an EOS. What the user must do for implementing a new thermodynamic model, is to construct new objects inherited from the `Interaction` and `FluidPhase` objects. A typical inherited `FluidPhase` object uses one or more different `Interaction` objects. It is worth noting that the only significant amount of programming work is the construction of parse trees. A large number of variables and functions are predefined in the `Interaction` and `FluidPhase` objects that the user need not recreate. The most important of these functions involve:

- the calculation of chemical potentials, fugacities, and activities, and their first-order differentiates with respect to the temperature, molar volume, pressure, and mole fraction;
- the calculation of the molar volume, given the temperature and the pressure;
- the calculation of liquid–gas, liquid–liquid, and liquid–liquid–gas equilibria (Asselineau, Bogdanic, and Vidal, 1979), including the calculation of various phase diagrams (isotherms, isobars, and isopleths for binary and ternary systems);
- the calculation of the critical curve of binary fluids.

APPLICATION TO THE ANDERKO AND PITZER EQUATION OF STATE

As an illustration of the usefulness of this library, the Anderko and Pitzer EOS (1993a, 1993b) has been chosen. It applies to the H_2O–NaCl–KCl system above 300°C. This ternary system is representative of fluids that are met in a wide variety of geological environments. In particular, near-critical conditions of this system are important for the understanding of boiling seafloor geothermal systems and of many hydrothermal deposits (Bischoff, Rosenbauer, and

Table 2. List of available objects in library for user. a, b, c and d stand for scalars involved in arithmetic expressions. u and v are used for "children" subexpressions

Object	Arithmetic expresion
Cube	$a\ (u + b)^3$
Exponential	$a\ \exp\ (bu\ +\ c)_d$
Inverse	$a/(u + b)$
Inverse2	$a/(u + b)^2$
Logarithm	$a\ \log\ (u + b)$
Power	$a(u + b)^c$
Powo	$a(u + b)^c$ with $c = n/m$ and m is an odd integer
Square	$a/(u + b)^2$
Sum1	$au + b$
Sum2	$au + bv + c$
Sum	$\Sigma a_i u_i$
Division	au/v
Product	auv
LineaX2	$\Sigma\Sigma a_{ij} x_i x_j$
LineaX3	$\Sigma\Sigma\Sigma a_{ijk} x_i x_j x_k$
SumX2	$\Sigma\Sigma x_i x_j u_{ij}$
SumX3	$\Sigma\Sigma\Sigma x_i x_j x_k u_{ijk}$
SumX4	$\Sigma\Sigma\Sigma\Sigma x_i x_j x_k x_l u_{ijkl}$
SumX5	$\Sigma\Sigma\Sigma\Sigma\Sigma x_i x_j x_k x_l x_m u_{ijklm}$

Pitzer, 1986). Our library has been used for solving two difficult problems:

- calculation of the critical locus curve of the H_2O–NaCl system,
- calculation of thermodynamic properties, requiring differentiation of the Helmholz free energy with respect to the temperature, such as the enthalpy, and others.

Calculation of derivatives of the Helmholtz free energy

The Anderko and Pitzer EOS considers the H_2O–NaCl–KCl system above 573 K as a mixture of H_2O molecules and the ion pairs NaCl and KCl. The Helmholtz free energy of H_2O–salts mixtures is split into three contributions:

$$A = A_{\text{repulsion}} + A_{\text{dipole}} + A_{\text{dispersion}}\ , \tag{5}$$

where $A_{\text{repulsion}}$ is a result of the repulsion between molecules and ion pairs, A_{dipole} is representative of dipole–dipole interactions, and $A_{\text{dispersion}}$ stands for the attractive dispersion forces. For programming this EOS, three separate `Interaction`—type objects have been defined for the intermolecular repulsion, dipolar, and dispersion forces. The source code for the repulsive part is given in Appendix D. Appendix E gives the source code for the Anderko and Pitzer EOS (1993). Some information about the construction of the parse trees of these three interactions is summarized here.

The repulsion contribution. The arithmetic expressions for the evaluation of $A_{\text{repulsion}}$ are given by Anderko and Pitzer (1993a, b) following the theory of hard spheres (Boublik, 1970; Mansoori and others, 1971):

$$\frac{A^{\text{rep}}}{RT} = \frac{\frac{3DE}{F}\eta - \frac{E^3}{F^2}}{1-\eta} + \frac{\frac{E^3}{F^2}}{(1-\eta)^2} + \left(\frac{E^3}{F^2} - 1\right)\ln(1-\eta)\ , \tag{6}$$

where η is the reduced density,

$$\eta = \frac{b}{4V}\ , \tag{7}$$

and b is the covolume of the mixture,

$$b = \sum_i x_i b_i\ , \tag{8}$$

where x_i is the mole fraction of the ith component and b_i is the covolume.

The b_i covolumes and the σ_i hard sphere diameters are related by:

$$b_i = \frac{2}{3}\pi N_A \sigma_i^3\ , \tag{9}$$

where N_A is the Avogadro number.

The parameters D, E, and F are given by:

$$D = \sum_i x_i \sigma_i \tag{10a}$$

$$E = \sum_i x_i \sigma_i^2 \tag{10b}$$

$$F = \sum_i x_i \sigma_i^3\ . \tag{10c}$$

The procedure for calculating the differentiates of the $A_{\text{repulsion}}$ has been completely developed in Appendix D. Figure 3 can be used as a guide for following the construction of the parse representation.

The dipole contribution. The Helmholtz energy difference between dipolar hard spheres and simple hard spheres is expressed as (Stell, Rasaiah, and Narang, 1972; Rushbrooke, Stell, and Høye, 1973; Stell, Rasaiah, and Narang, 1974):

$$\frac{A^{\text{dip}}}{RT} = \frac{A_2}{1 - \frac{A_3}{A_2}}\ , \tag{11a}$$

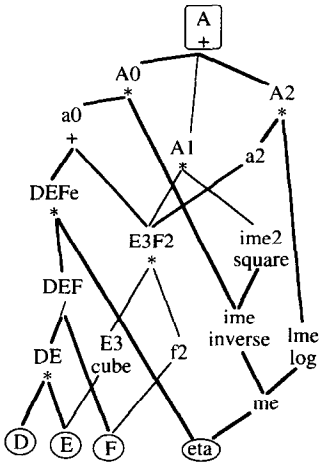

Fig. 3. Parse tree of Helmholz free energy for a mixture of hard spheres, following theory of Boublik (Boublik, 1970; Anderko and Pitzer, 1993a, 1993b). Leaves are surrounded by circles, and root is surrounded by rectangular frame. Function performed by each node is indicated by symbol below name of each subexpression. Notations are those used in declaration of `Boublik` object (see Appendix D).

where

$$A_2 = K_2 \eta I_2(\eta) \tag{11b}$$

$$A_3 = K_3 \eta^2 I_3(\eta)\,, \tag{11c}$$

$I_2(\eta)$ and $I_3(\eta)$ are empirical functions of the η density. Auxiliary functions, termed K_2 and K_3 have been introduced in order to simplify expressions:

$$K_2 = \sum_i \sum_j k_{2,ij} x_i x_j\,, \tag{12a}$$

where

$$k_{2,ij} = -\frac{4}{3}\left(\frac{\sigma_i^3 \sigma_j^3}{\sigma_{ij}^6}\right)\frac{\eta_{ij}}{\eta}\tilde{\mu}_i^2 \tilde{\mu}_j^2\,, \tag{12b}$$

and

$$K_3 = \sum_i \sum_j \sum_k k_{3,ijk} x_i x_j x_k\,, \tag{13a}$$

where

$$k_{3,ijk} = \frac{10}{9}\left(\frac{\sigma_i^3 \sigma_j^3 \sigma_k^3}{\sigma_{ijk}^9}\right)\frac{\eta_{ijk}^2}{\eta^2}\tilde{\mu}_i^2 \tilde{\mu}_j^2 \mu \tilde{\mu}_k^2\,. \tag{13b}$$

η_{ij} and η_{ijk} are functions of the density and of the composition of the mixture, $\tilde{\mu}_i$ is a function of the dipole moment of the ith component, K_2 is a quadratic function of the composition, and K_3 is a cubic function of the composition. For handling these finite summations, `LineaX2` and `LineaX3` objects of the library have been used.

The dispersion contribution.

$$\frac{A^{\text{dis}}}{RT} = -\frac{1}{RT}\left(\frac{a}{V} + \frac{acb}{4V^2} + \frac{adb^2}{16V^3} + \frac{aeb^3}{64V^4}\right) \tag{14}$$

where

$$b = \sum_i x_i b_i \tag{15}$$

$$a = \sum_i \sum_j x_i x_j a_{ij} \tag{16}$$

$$c = \frac{1}{ab}\sum_i \sum_j \sum_k x_i x_j x_k (ac)_{ijk} b_{ijk} \tag{17}$$

$$d = \frac{1}{ab^2}\sum_i \sum_j \sum_k \sum_l x_i x_j x_k x_l (ad)_{ijkl} b_{ijkl}^2 \tag{18}$$

$$e = \frac{1}{ab^3}\sum_i \sum_j \sum_k \sum_l \sum_m x_i x_j x_k x_l x_m (ae)_{ijklm} b_{ijklm}^3\,. \tag{19}$$

Calculation of differentiates of the a, c, d and e summations requires, respectively, the use of `SumX2`, `SumX3`, `SumX4`, and `SumX5` objects, as explained in Appendix C.

Calculation of the critical curve

A binary mixture of components 1 and 2 is in a critical state if the two following relations S_1 and S_2 are equal to zero (Heilig, 1988):

$$S_1(T,V,x_2) = A_{x_2 x_2} A_{VV} - A_{V x_2} = 0 \tag{20a}$$

$$S_2(T,V,x_2) = A_{VV}^2 A_{x_2 x_2 x_2} - 3A_{VV} A_{V x_2} A_{V x_2 x_2} + 3A_{V V x_2} A_{V x_2}^2 - A_{VVV} A_{x_2 x_2} A_{V x_2} = 0\,, \tag{20b}$$

where A is the free Helmholtz energy of the mixture, as given by the EOS. This set of non-linear equations has been solved by the Newton method with the x_2 mole fraction as a given parameter, and the temperature T and the molar volume V as unknowns.

Once a solution has been obtained, it is still necessary to check the stability of this critical point by considering the following condition (Heilig, 1988; Heilig and Franck, 1990):

$$A_{x_2 x_2 x_2 x_2} - 4qA_{V x_2 x_2 x_2} + 6q^2 A_{VV x_2 x_2} - 4q^3 A_{VVV x_2} + q^4 A_{VVVV} \geq 0\,, \tag{21a}$$

$$\text{with } q = \frac{A_{V x_2}}{A_{VV}}\,. \tag{21b}$$

Thus, derivatives up to the fourth order with respect to the temperature, the molar volume, and the composition are required and are calculated with our library. When a critical point has been determined, it is possible to estimate the variation of critical temperature and molar volume for a small change of the composition. This possibility is interesting, as it allows the estimation of suitable values of T and V for the Newton algorithm of the next critical point on

the critical curve. The solution of the following linear system is required:

$$\begin{bmatrix} \left(\frac{\partial S_1}{\partial T}\right)_{V,x_2} & \left(\frac{\partial S_1}{\partial V}\right)_{T,x_2} \\ \left(\frac{\partial S_2}{\partial T}\right)_{V,x_2} & \left(\frac{\partial S_2}{\partial V}\right)_{T,x_2} \end{bmatrix} \begin{bmatrix} \left(\frac{\partial T}{\partial x_2}\right)_{S_1,S_2} \\ \left(\frac{\partial V}{\partial x_2}\right)_{S_1,S_2} \end{bmatrix} = \begin{bmatrix} -\left(\frac{\partial S_1}{\partial x_2}\right)_{T,V} \\ \left(\frac{\partial S_2}{\partial x_2}\right)_{T,V} \end{bmatrix} . \quad (22)$$

This result is obtained by writing the Taylor series of S_1 and S_2 functions along the critical line, truncated after the first term:

$$dS_1 = 0 = \left(\frac{\partial S_1}{\partial T}\right)_{V,x_2} dT + \left(\frac{\partial S_1}{\partial V}\right)_{T,x_2} dV + \left(\frac{\partial S_1}{\partial x_2}\right)_{T,V} dx_2 , \quad (23a)$$

$$dS_2 = 0 = \left(\frac{\partial S_2}{\partial T}\right)_{V,x_2} dT + \left(\frac{\partial S_2}{\partial V}\right)_{T,x_2} dV + \left(\frac{\partial S_2}{\partial x_2}\right)_{T,V} dx_2 . \quad (23b)$$

It is worth noting that the left-hand matrix (Eq. 22) is calculated for the solving the nonlinear system (Eq. 20) with the Newton algorithm. Thus, this method permits a fast and stepwise calculation of the critical line of a binary mixture. It is particularly valuable for water–salts systems, where a small increment of the $x_2 (= x_{NaCl})$ mole fraction value results in a large variation of critical P and T variables near the H_2O critical point. $T - x_{NaCl}$ and $P - T$ projections of the critical curve of the H_2O–NaCl system calculated with the Anderko and Pitzer EOS (1993a) are given in Figure 4.

Calculation of the isobaric expansivity

The isobaric expansivity (or thermal expansion coefficient) exemplifies an important thermodynamic parameter, which is difficult to extract from the Anderko and Pitzer EOS (1993a, 1993b), as its calculation involves a temperature differentiation of the Helmholtz free energy. This parameter is required for calculating the Rayleigh–Darcy number, which is a key parameter in modelling fluid circulations in the crust (Johnson and Norton, 1991), in particular for locating the transition between conductive and convective regimes. The expansion coefficient is defined as:

$$\alpha = \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_p . \quad (25)$$

$(\partial V/\partial T)_p$ can be obtained from the Helmholtz free energy by noting that:

$$\left(\frac{\partial V}{\partial T}\right)_p = \frac{\left(\frac{\partial P}{\partial T}\right)_V}{\left(\frac{\partial P}{\partial V}\right)_T} = -\frac{\left(\frac{\partial^2 A}{\partial T \partial V}\right)}{\left(\frac{\partial^2 A}{\partial V^2}\right)_T} . \quad (26)$$

Illustration of the variation of the isobaric expansivity as a function of the temperature and the NaCl mole fraction is given in Figure 5.

CONCLUSION

This library is used currently in our laboratory and it can be applied to a large variety of theoretical and applied geochemical problems, including, for example:

- Phase topology studies of fluid mixtures, where derivatives up to the seventh order are sometimes required, whatever the complexity of the EOS (Deiters and Pegg, 1989; Van Pelt and de Loos, 1992; Thiéry and Lvov, 1996).
- Characterization of fluid inclusions where critical and retrograde condensation conditions can be determined accurately with models developed for CO_2–CH_4–N_2 fluid inclusions (Thiéry, Vidal, and Dubessy, 1994; Thiéry, Kerkhof, and Dubessy, 1994; Thiéry and Dubessy, 1996).

Advantages of this library are numerous. First, it is reliable. Checking functions are supplied which allow analytical and numerical values to be compared. Detection of mistakes is easy, as each node of the parse graph can be checked individually. Secondly, it is a versatile tool. In research, it is often necessary to test between several hypotheses. With such a library, it is straightforward, for instance, to modify the temperature law for one of the parameters of the EOS, or to modify a mixing rule. Without this library, an extensive reprogramming of the computer code would be required in some situations. Next, this library is able to take into account complex operators such as simple finite summation (Σ), double summation ($\Sigma\Sigma$), etc., up to the quintuple summation ($\Sigma\Sigma\Sigma\Sigma\Sigma$). These operators are, indeed, widespread in modelling fluid properties. Finally, this library is an efficient and helpful tool for programming thermodynamic models. Before, one week of painstaking code development and testing was necessary to differentiate the Helmholtz free energy function, such as the one proposed for hard spheres mixtures (Boublik, 1970), which can now be achieved in two hours. Moreover, when combined with our library of thermodynamic calculations, any complex thermodynamic function is derived automatically. Another point of interest is that programs that implement an EOS based on this library will share the same routines and interface. This is an important consideration for implementing and updating calculation programs.

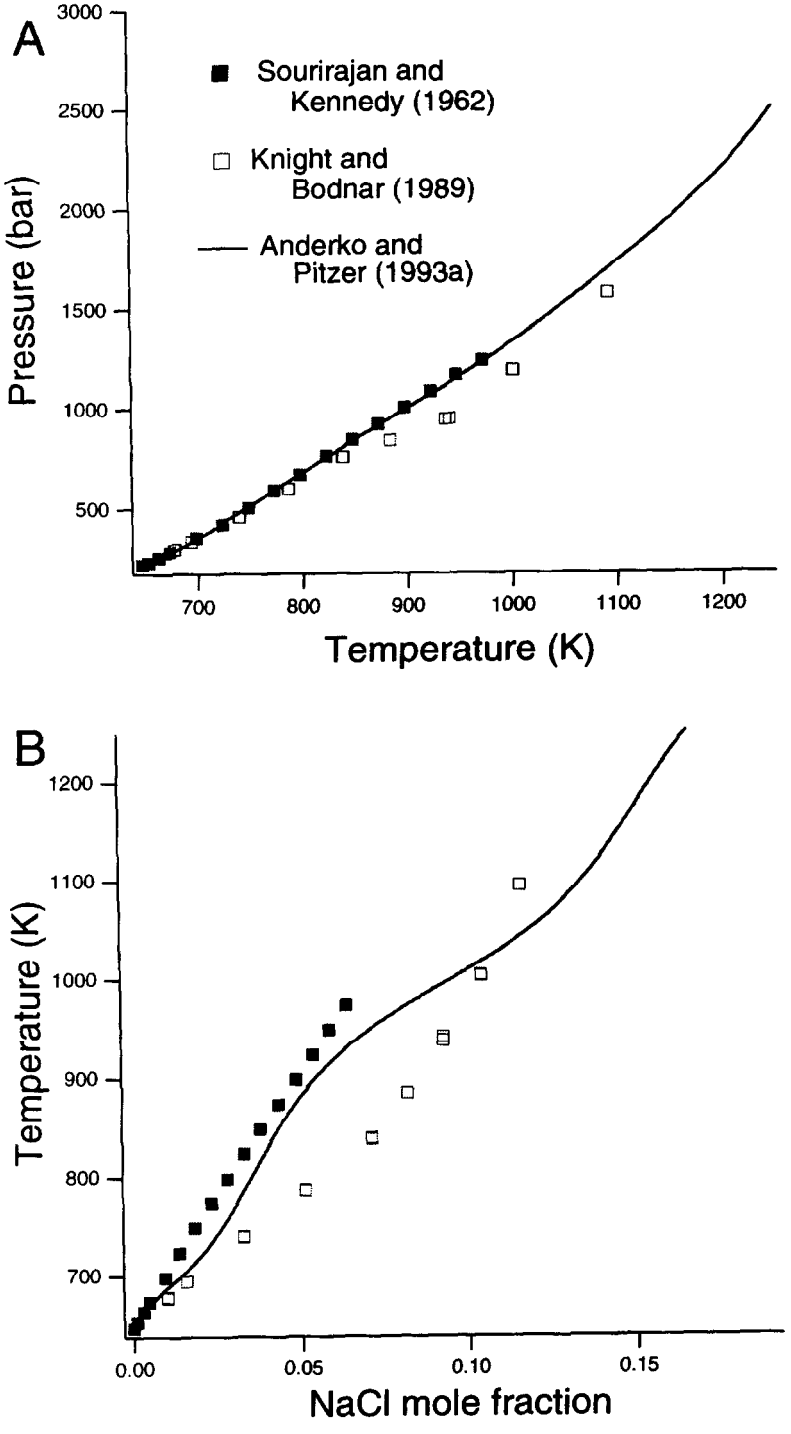

Figure 4. H_2O–NaCl critical line in P–T diagram (A), and T–X NaCl diagram (B) calculated by Anderko and Pitzer EOS with aid of this library. Experimental data are those of Sourirajan and Kennedy (1962), and Knight and Bodnar (1989).

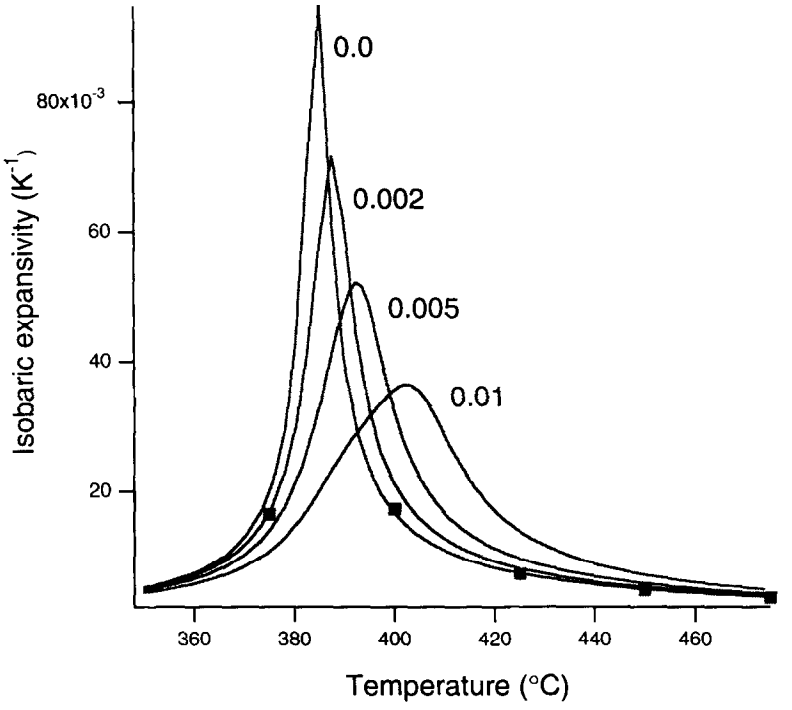

Figure 5. Isobaric expansivity (K^{-1}) as function of temperature and NaCl mole fraction. Curves have been calculated with Anderko and Pitzer (1993a) EOS for a pressure of 250 bars, that is above critical pressure of H_2O (P_c = 217.6bar). Numbers indicate NaCl mole fraction of aqueous solution. Full squares represent isobaric expansivity calculated for pure H_2O fluid with Haar, Gallagher and Kell EOS (1984), reported by Johnson and Norton (1991).

Note. The source code of the library and detailed examples are available by anonymous FTP from the server IAMG.ORG, or from the Web at http://www.iamg.org.

Acknowledgments—This work has been supported financially by CNRS and Elf Aquitaine Production. The paper has benefited greatly from fruitful discussions and careful reviews of earlier drafts with Christophe Monnin, Ronald Bakker and Jean Dubessy. The two anonymous reviewers are thanked for their comments and suggestions for improving the manuscript.

REFERENCES

Angus, S., Armstrong, B., and de Reuck, K. M., 1976, Carbon dioxide: International Thermodynamic Tables of the Fluid State, v. 3, Pergamon Press, Oxford, 385 p.

Angus, S., Armstrong, B., and de Reuck, K. M., 1978, Methane: International Thermodynamic Tables of the Fluid State, v. 5, Pergamon Press, Oxford, 251 p.

Anderko, A., and Pitzer, K. S., 1993, Equation-of-state representation of phase equilibria and volumetric properties of the system NaCl-H_2O above 573 K: Geochim. Cosmochim. Acta, v. 57, no. 8, p. 1657–1680.

Anderko, A., and Pitzer, K. S., 1993, Phase equilibria and volumetric properties of the systems KCl-H_2O and NaCl-KCl-H_2O above 573 K, Equation of state representation: Geochim. Cosmochim. Acta, v. 57, no. 20, p. 4885–4897.

Asselineau, L., Bogdanic, G., and Vidal, J., 1979, A versatile algorithm for calculating vapour–liquid equilibria: Fluid Phase Equilibria, v. 3, p. 273–290.

Baker, L. E., and Luks, D., 1978, Critical point and saturation pressure calculations for multicomponent systems: Soc. Petrol. Eng. Jour., p. 1–9.

Bischoff, J. L., Rosenbauer, R. J., and Pitzer, K. S., 1986, The system NaCl–H_2O: relations of vapor–liquid near the critical temperature of water and of vapor–liquid–halite from 300 to 500°C: Geochim. Cosmochim. Acta, v. 50, no. 7, p. 1437–1444.

Boublik, T., 1970, Hard sphere equation of state: Jour. Chem. Phys., v. 53, no. 1, p. 471–472.

Chen, D., and Moler, C., 1993, Symbolic math toolbox user's guide: The MathWorks, Inc., Natick, Massachusetts, 166 p.

Deiters, U. K., and Pegg, I. L., 1989, Systematic investigation of the phase behavior in binary fluid mixtures. 1. Calculations based on the Redlich-Kwong equation of state: Jour. Chem. Phys., v. 90, no. 11, p. 6632–6641.

Eaton, B. E., 1988, On the calculation of critical points by the method of Heidemann and Khalil: National Bureau of Standards, Technical Note 1313, 52 p.

Haar, L., Gallagher, J. S., and Kell, G. S., 1984, NBS/NRC steam tables: thermodynamic and transport properties and computer programs for vapor and liquid of water in SI Units: Hemisphere, New York, 320 p.

Harvey, A. H., 1991, Phase equilibria and critical lines in model water/salt mixtures: Jour. Chem. Phys., v. 95, no. 1, p. 479–484.

Haugeland, J., 1985, Artificial intelligence, the very idea: Massachusetts Institute of Technology, Cambridge, Massachusetts, 345 p.

Hearn, A. C., 1987, Reduce users manual version 3.3: The Rand Corporation, Santa Monica, California, 243 p.

Heidemann, R. A., and Khalil, A. M., 1980, The calculation of critical points: Am. Inst. Chem. Eng. Jour., v. 26, p. 769–779.

Heilig, M., 1988, Berechnung und experimentelle bestimmung von zustandsdiagrammen fluider mehrkomponen-

tensysteme bis 400°C und 2500 bar: Unpubl. Ph.D. dissertation, Univ. Karlsruhe, Karlsruhe, Germany, 143 p.
Heilig, M., and Franck, E. U., 1990, Phase equilibria of multicomponent fluid systems to high pressures and temperatures: Ber. Bunsenges. Phys. Chem., v. 94, p. 27–35.
Hill, P. G., 1990, A unified fundamental equation for the thermodynamic properties of H_2O: Jour. Phys. Chem. Ref. Data, v. 19, no. 5, p. 1233–1274.
Huron, M. J., Dufour, G.-N., and Vidal, J., 1977, Vapour–liquid equilibrium and critical locus curve calculations with the Soave equation for hydrocarbons systems with carbon dioxide and hydrogen sulphide: Fluid Phase Equilibria, v. 1, p. 247–265.
Johnson, J. W., and Norton, D., 1991, Critical phenomena in hydrothermal systems: state, thermodynamic, electrostatic and transport properties of H_2O in the critical region, Am. Jour. Sci., v. 291, p. 541–648.
Keenan J. H., Keyes, F. G., Hill, P. G., and Moore, J. G., 1969, Steam tables: thermodynamic properties of water including vapor, liquid and solid phases (English units): John Wiley and Sons, New York, 162 p.
Kernighan, B. W., and Ritchie, D. M., 1978, The C programming language: Prentice-Hall, Englewood Cliffs, New Jersey, 228 p.
Knight, C. L., and Bodnar, R. J., 1989, Synthetic fluid inclusions: IX. Critical *PVTX* properties of NaCl–H_2O solutions: Geochim. Cosmochim. Acta, v. 53, no. 1, p. 3–8.
Lvov, S. N., and Wood, R. H., 1990, Equations of state of aqueous NaCl solutions over a wide range of temperatures, pressures and concentrations: Fluid Phase Equilibria, v. 60, p. 273–287.
Mansoori, G. A., Carnahan, N. F., Starling, K. E., and Leland, T. W., 1971, Equilibrium thermodynamics of the mixture of hard-spheres: Jour. Chem. Phys., v. 54, no. 4, p. 1523–1525.
Michelsen, M. L., 1980, Calculation of phase envelopes and critical points for multicomponent mixtures: Fluid Phase Equilibria, v. 4, p. 1–10.
Michelsen, M. L., and Heidemann, R. A., 1981, Calculation of critical points from cubic two-constant equations of state: Am. Inst. Chem. Eng. Jour., v. 27, p. 521–523.
Norris, A. C., 1981, Computational chemistry. an introduction to numerical methods: John Wiley and Sons Inc., New York, 454 p.
Peng, D. Y., and Robinson, D. E., 1977, A rigorous method for predicting the critical properties of multicomponent systems from an equation of state: Am. Inst. Chem. Eng. Jour., v. 23, p. 137–144.
Rushbrooke, G. S., Stell, G. and Høye, J. S., 1973, Theory of polar fluids I. Dipolar hard spheres: Jour. Molecular Phys., v. 26, p. 1199–1215.
Sedgewick, R., 1992, Algorithms in C++: Princeton University, Addison-Wesley Publ. Co., Reading, Massachusetts, 656 p.
Sourirajan, S., and Kennedy, G. C., 1962, The system H_2O–NaCl at elevated temperatures and pressures: Am. Jour. Sci., v. 260, p. 115–141.
Stell, G., Rasaiah, J. C., and Narang, H., 1972, Thermodynamic pertubation theory for simple polar fluids: Jour. Molecular Phys., v. 23, p. 393–406.
Stell, G., Rasaiah, J. C., and Narang, H., 1974, Thermodynamic pertubation theory for simple polar fluids II: Jour. Molecular Phys., v. 27, p. 1393–1414.
Stroustrup, B., 1991, The C++ programming language (2nd ed.): Addison-Wesley Publ. Co., Reading, Massachusetts, 284 p.
Thiéry, R., Vidal, J., and Dubessy, J., 1994, Phase equilibria modelling applied to fluid inclusions. Liquid–vapour equilibria and calculation of the molar volume in the CO_2–CH_4–N_2 system: Geochim. Cosmochim. Acta, v. 58, no. 3, p. 1073–1082.
Thiéry, R., Kerkhof van den, A. M., and Dubessy, J., 1994, *vX* properties modelling of the CO_2–CH_4 and CO_2–N_2 fluids up to 31°C: Eur. Jour. Mineral., v. 6, p. 753–771.
Thiéry, R., and Dubessy, J., 1996, Improved liquid–vapour phase equilibria modelling in the CO_2–CH_4–N_2 system for critical and noncritical conditions: Fluid Phase Equilibria (in press).
Thiéry, R., and Lvov, S. N., 1996, The ion-dipole model based on the mean spherical approximation. I. the global phase diagram: Jour. Chem. Phys. (submitted).
Van Pelt, A., and de Loos, Th. W., 1992, Connectivity of critical lines around the van Laar point in *T* X projections: Jour. Chem. Phys., v. 97, no. 2, p. 1271–1281.
Wolfram, S., 1991, Mathematica, a system for doing mathematics by computer (2nd ed.): Addison-Wesley Publ. Co., Redwood City, California, 1056 p.

APPENDIX A

For constructing the parse representation of a sum, such as:

$$P = P_1 + 2P_2 - P_3 + 5P_4 + 3,$$

the following procedure must be used:

```
//-------------------------------------------
//  Use of the Sum object
//-------------------------------------------

      // the declarations of nodes

Sum P ;
Leaf P1, P2, P3, P4 ;

      // Instructions to insert at the
      // beginning of the program

P . number_of_terms (4) ;

P . tie ( & P1, & P2 , & P3 , & P4 ) ;
P . set_scalars ( 1.0 , 2.0 , -1.0 , 5.0 , 3.0 ) ;
```

APPENDIX B

For constructing the parse representations of expressions such as:

$$K_2 = \sum_i \sum_j x_i x_j k_{2,ij}, \text{ and}$$

$$K_3 = \sum_i \sum_j \sum_k x_i x_j x_k k_{3,ijk},$$

where $k_{2,ij}$ and $k_{2,ijk}$ are scalars, the following instructions must be written:

```
//-------------------------------------------
// Use of the LineaX2 and Lineax3 objects
//-------------------------------------------

      // the declarations of variables

int N = 5 ;       // the number of components
int ix = 2 ;      // the index of the first composition
                  // variable

double ** k2, *** k3 ; // the array of scalars

LineaX2 K2 ;
LineaX3 K3 ;


      // Instructions to insert at the
      // beginning of the program

K2 . dimension (N) ;
K3 . dimension (N) ;

K2 . starting_index (ix) ;
K3 . starting_index (ix) ;
```

```
k2 = sym2alloc (N) ;    // sym2alloc and sym3alloc ()
k3 = sym3alloc (N) ;    // functions allocates symmetrical
                        // arrays of scalars

K2 . set_scalars (k2) ;
K3 . set_scalars (k3) ;


      // Instructions to be called before
      // any calculation of differentiates

           // x is the array of mole fractions

K2 . set_xvariables ( x ) ;
K3 . set_xvariables ( x ) ;
```

APPENDIX C

```
//-------------------------------------------------------------
// Use of SumX2, SumX3, SumX4 and SumX5 objects
//-------------------------------------------------------------

     // the declarations of variables

int N = 5 ;       // the number of components
int ix = 2 ;      // the index of the first composition
                  // variable

SumX2 a ;         // the virial expansions
SumX3 c ;
SumX4 d ;
SumX5 e ;

Product ***    aij   ;      // These are the array of the nodes
Product ****   cijk  ;
Product *****  dijkl ;
Product ****** eijklm ;


a . number_of_xvariables ( N ) ;   // Instructions to insert at the
c . number_of_xvariables ( N ) ;   // beginning of the program
d . number_of_xvariables ( N ) ;


e . number_of_xvariables ( N ) ;

b . starting_index (ix) ;
a . starting_index (ix) ;
c . starting_index (ix) ;
d . starting_index (ix) ;
e . starting_index (ix) ;

b . tie ( bi     ) ;
a . tie ( aij    ) ;
c . tie ( cijk   ) ;
d . tie ( dijkl  ) ;
e . tie ( eijklm ) ;


     // Instructions to be called before
     // any calculation of differentiates

a . set_xvariables ( x ) ;   // x is the array of mole fractions
c . set_xvariables ( x ) ;
d . set_xvariables ( x ) ;
e . set_xvariables ( x ) ;
```

APPENDIX D

```
#ifndef _BOUBLIK_H
#define _BOUBLIK_H

//-------------------------------------------------------------
//
//    Header file : Boublik.h
//
//    Contents : the description of the Boublik object
//
//-------------------------------------------------------------


#include "Interaction.h"
#include "Logarithm.h"
#include "Inverse.h"
#include "Inverse2.h"
#include "Product.h"
#include "Cube.h"
#include "Square.h"
#include "Division.h"
#include "sum.h"
#include "Leaf.h"
#include "util.h"


//-------------------------------------------------------------
     class Boublik : public Interaction
//-------------------------------------------------------------
//
//    The hard-sphere fluid with the Boublik-Mansoori
//    mixing rules
//
//-------------------------------------------------------------

     {
public :

     Leaf D ;                  // dimensionless parameters
     Leaf E ;
     Leaf F ;

     double b      ;           // covolumes
     double * bi   ;
     double * sig  ;           // hard sphere diameters

     Leaf       eta   ;
     Sum1       me    ;             // = 1.0 - eta
     Inverse    ime   ;             // = 1.0 / me
     Square     ime2  ;             // = 1.0 / ( me * me )
     Logarithm  lme   ;             // = log ( me )

     Product    DE    ;             // = D * E
     Inverse2   f2    ;             // = 1.0 / ( F * F )
     Cube       E3    ;             // = E * E * E
     Division   DEF   ;             // = DE / F
     Product    E3F2  ;             // = E3 * f2
     Product    DEFe  ;             // = DEF * eta

     Sum2       a0    ;             // 3.0 * DEFe - E3F2
     Sum1       a2    ;             // E3F2 - 1

     Leaf       mlogV ;             // = - log (V )
     Product    A0    ;             // = a0   * ime
     Product    A1    ;             // = E3F2 * ime2
     Product    A2    ;             // = a2   * lme


     Boublik () ;
     void number_of_components (int) ;
     void set_hard_sphere_diameters ( double , ... ) ;

     void calculate_fixed_parameters () ;
     void calculate_volume_dependent_parameters () ;
     void calculate_high_order_parameters () ;

     void check_derivatives  () ;
     } ;

#endif

#include "Boublik.h"
#include <stdarg.h>

#define iT  0
#define iV  1
#define ix  2
#define i1  2
#define i2  3


//-------------------------------------------------------------
     Boublik :: Boublik () : Interaction ()
//-------------------------------------------------------------
{
     //-------------------------------------------
     //    Construction of the parse tree
     //-------------------------------------------

                                   //    A = A0 + A1 + A2
A = new Sum ;


     ((Sum*) A) -> number_of_terms (4) ;
     ((Sum*) A) -> tie ( &mlogV , & A0 , & A1 , & A2 ) ;
     ((Sum*) A) -> set_scalars ( 1.0 , 1.0 , 1.0 , 1.0 , 0.0 ) ;

     A0   . tie ( a0 , ime ) ;          //    A0 = a0 * ime
     A1   . tie ( E3F2 , ime2 ) ;       //    A1 = E3F2 * ime2
     A2   . tie ( a2 , lme ) ;          //    A2 = a2 * lme

                                        //    a0 = 3.0 * DEFe - E3F2 + 0.0
     a0   . tie ( DEFe , E3F2 ) ;
     a0   . set_scalars ( 3.0 , - 1.0 , 0.0 ) ;

                                        //    a2 = E3F2 - 1.0
     a2   . tie ( E3F2 ) ;
     a2   . set_scalars ( 1.0 , - 1.0 ) ;

     DEFe . tie ( DEF , eta ) ;         //    DEFe = DEF * eta
     E3F2 . tie ( E3 , f2 ) ;           //    E3F2 = E3 * f2
     DEF  . tie ( DE , F ) ;            //    DEF = DE / F
     E3   . tie ( E ) ;                 //    E3 = cube (E)
     f2   . tie ( F ) ;                 //    f2 = 1 / ( F * F ) ;
     DE   . tie ( D , E ) ;             //    DE = D * E

                                        //    me = - eta + 1.0
     me   . tie ( eta ) ;
     me   . set_scalars ( - 1.0 , 1.0 ) ;

     ime  . tie ( me  ) ;               //    ime  = 1.0 / me
     ime2 . tie ( ime ) ;               //    ime2 = 1.0 / ( me * me )
     lme  . tie ( me  ) ;               //    lme = log ( me )

     //-------------------------------------------
     //    Used for checking functions check1(),
     //    check2(), check3() and check4()
     //-------------------------------------------
```

continued

```
((Sum*) A) -> set_relations_with ( this ) ;
}

//-------------------------------------------------------------------
      void Boublik :: number_of_components ( int ncomp )
//-------------------------------------------------------------------
{
Interaction :: number_of_components (ncomp) ;
sig = allocate (N) ;
bi  = allocate (N) ;

A    -> number_of_variables ( 2+N ) ;

      //-------------------------------------------------------------
      // Gives useful indications of the dependency
      // on the variables
      //-------------------------------------------------------------

            //    For the leaves

mlogV . specify_variable_dependency ( iT , 0 ) ;
eta   . specify_variable_dependency ( iT , 0 ) ;
D     . specify_variable_dependency ( iT , 0 ) ;
E     . specify_variable_dependency ( iT , 0 ) ;
F     . specify_variable_dependency ( iT , 0 ) ;


D     . specify_variable_dependency ( iV , 0 ) ;
E     . specify_variable_dependency ( iV , 0 ) ;
F     . specify_variable_dependency ( iV , 0 ) ;

for ( int i = 0 ; i < N ; i ++ )
      mlogV . specify_variable_dependency ( ix+i , 0 ) ;

A   -> analyze_dependencies ( ) ;
}

//-------------------------------------------------------------------
      void Boublik :: calculate_fixed_parameters ()
//-------------------------------------------------------------------
//
//    Calculates parameters which are volume-independent.
//
//-------------------------------------------------------------------
{
RT = R * T ;

int i ;
double sigi, Dval, Eval, Fval ;

b = Dval = Eval = Fval = 0.0 ;

for ( i = 0 ; i < N ; i ++ )
      {
      b += x [i] * bi  [i] ;
      sigi = sig [i] ;

      Dval += x [i] * sigi ;
      Eval += x [i] * sigi * sigi ;
      Fval += x [i] * sigi * sigi * sigi ;
      }

D () = Dval ;
E () = Eval ;
F () = Fval ;
}

//-------------------------------------------------------------------
  void Boublik :: calculate_volume_dependent_parameters ()
//-------------------------------------------------------------------
{
int i ;
double sigi ;
double V2, V3, V4 ;


V2 = V  * V ;
V3 = V2 * V ;
V4 = V3 * V ;

mlogV () = - log ( V ) ;
mlogV (iV) = - 1 / V ;
mlogV (iV,iV) = 1.0 / V2 ;

for ( i = 0 ; i < N ; i ++ )
      {
      sigi = sig [i] ;

      eta (ix+i)           =          bi [i] / ( 4.0 * V  ) ;

      eta (ix+i,iV)        =       - bi [i] / ( 4.0 * V2 ) ;
      eta (ix+i,iV,iV)     =         bi [i] / ( 2.0 * V3 ) ;
      eta (ix+i,iV,iV,iV) = - 3.0 * bi [i] / ( 2.0 * V4 ) ;

      D (ix+i) = sigi ;             //    Derivatives wrt to composition
      E (ix+i) = sigi * sigi ;
      F (ix+i) = sigi * sigi * sigi ;
                          //    derivatives of higher order are zero
      }

eta ()           =   b / ( 4.0 * V  ) ;
eta (iV)         = - b / ( 4.0 * V2 ) ;
eta (iV,iV)      =   b / ( 2.0 * V3 ) ;
}

//-------------------------------------------------------------------
      void Boublik :: calculate_high_order_parameters ()
//-------------------------------------------------------------------
{
int i ;
double sigi ;
double V2, V3, V4, V5 ;

V2 = V  * V ;
V3 = V2 * V ;
V4 = V3 * V ;
V5 = V4 * V ;

mlogV (iV,iV,iV) = - 2.0 / V3 ;
mlogV (iV,iV,iV,iV) = 6.0 / V4 ;

for ( i = 0 ; i < N ; i ++ )
      {
      sigi = sig [i] ;
      eta (ix+i,iV,iV)    =         bi [i] / ( 2.0 * V3 ) ;
      eta (ix+i,iV,iV,iV) = - 3.0 * bi [i] / ( 2.0 * V4 ) ;
```

APPENDIX E

```
#ifndef _ANDERKO_H
#define _ANDERKO_H

#include "FluidPhase.h"
#include "Interaction.h"
#include "NewtonRaphson.h"
#include "Newton.h"

//-------------------------------------------------------------------
      class Anderko : public FluidPhase
//-------------------------------------------------------------------
//    Modelling of the H2O-NaCl-KCl system
//-------------------------------------------------------------------
      {
public :
            // Different types of interactions

      Interaction * repulsion    ; // between hard spheres

      Interaction * dipole       ; // between dipoles
      Interaction * dispersion   ; // dispersion forces

            // Parameters of the Anderko & Pitzer's EOS
                  // 1 : H2O
                  // 2 : NaCl
                  // 3 : KCl

      double b1   ;     // Covolumes
      double b2   ;
      double b3   ;
      double mu1  ;     // Dipole moments
      double mu2  ;
      double mu3  ;

      Anderko (int) ;
      } ;
#endif


#include "Anderko.h"

#include "Boublik.h"
#include "Dipole.h"
#include "H2ONaClDisp.h"

//-------------------------------------------------------------------
  Anderko :: Anderko (int nphase) : FluidPhase (3,nphase)
//-------------------------------------------------------------------
{
interaction [0] = repulsion  = new Boublik ;
interaction [1] = dipole     = new Dipole  ;
interaction [2] = dispersion = new H2ONaClDisp ;
}
```