Volume 126, Article No. 126033 (2021) https://doi.org/10.6028/jres.126.033
Journal of Research of National Institute of Standards and Technology

Precise Numerical Differentiation of Thermodynamic Functions with Multicomplex Variables

Ulrich K. Deiters[1] **and Ian H. Bell**[2]

[1]Institute of Physical Chemistry, Faculty of Mathematics and Natural Sciences, University of Cologne, D-50939 Köln, Germany

[2]National Institute of Standards and Technology, Boulder, CO 80305, USA

ulrich.deiters@uni-koeln.de
ian.bell@nist.gov

The multicomplex finite-step method for numerical differentiation is an extension of the popular Squire–Trapp method, which uses complex arithmetics to compute first-order derivatives with almost machine precision. In contrast to this, the multicomplex method can be applied to higher-order derivatives. Furthermore, it can be applied to functions of more than one variable and obtain mixed derivatives. It is possible to compute various derivatives at the same time.
This work demonstrates the numerical differentiation with multicomplex variables for some thermodynamic problems. The method can be easily implemented into existing computer programs, applied to equations of state of arbitrary complexity, and achieves almost machine precision for the derivatives. Alternative methods based on complex integration are discussed, too.

Key words: multicomplex arithmetics; numerical differentiation.

Accepted: October 19, 2021

Published: November 29, 2021

https://doi.org/10.6028/jres.126.033

1. Introduction

Differentiation of functions plays a major role in many disciplines of science, from physics to engineering. A special case is thermodynamics, which is practically built upon differentiation: thermodynamic theories and models usually represent observable properties of matter as derivatives of some master functions, e.g., the Helmholtz energy or the Gibbs energy. For example, pressure is a derivative of the Helmholtz energy with respect to volume (at constant temperature), or chemical potentials are derivatives of master functions mentioned above with respect to amount of substance. In particular, stability criteria are often related to second-order derivatives of the master functions; for example, mechanical stability depends on the sign of the isothermal compressibility, which in turn is related to a second-order derivative of the Helmholtz energy with respect to volume.

Differentiation usually does not pose a principal problem, as most functions that one encounters in science or engineering are analytic. The calculation of derivatives can, however, pose a practical problem, as functions (with the exception of polynomials) tend to become more complicated upon differentiation and computationally more costly to evaluate. Differentiating the functions of complicated thermodynamic models “by hand” is therefore often neither a pleasant nor an economical task. An alternative are symbolic-algebra computer programs. These can not only perform differentiations of arbitrary complexity, but also—within limits—simplify the results. Such programs are, however, rather expensive and not generally available. Moreover, they may not be applicable if the operand function cannot be written down explicitly, for instance because it contains terms that require iterations.

Such equations of state for fluids do indeed exist:

- The equation of state of Dieterici [1] of 1899 was published in a pressure-explicit form; it cannot be integrated analytically. This equation is not obsolete: Sadus [2, 3] reported some favourable thermodynamic features in 2001. Polishuk and Vera predicted critical curves of mixtures, but had to use a truncated version of this equation, which could be integrated analytically.
- Boshkova and Deiters used temperature- and density-dependent hard-core volumes (obtained from perturbation theory by means of an iterative procedure) to extend equations of state to very high pressures [4].
- Equations of state comprising chemical association models almost always contain a variable that represents the fraction of not associated molecules or binding sites. Examples are the SAFT equation of state [5, 6] and the numerous models derived from it.

There exist program libraries for implicit symbolic differentiation, e.g., `autodiff` for C++ programs [7], but the integration of the code into existing programs is not always easy.

This leaves numerical differentiation methods. The most widely known ones are finite-difference methods like, e.g.,

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \frac{f(x+h)-f(x-h)}{2h} + \mathrm{O}(h^2)\,, \tag{1}$$

for first-order derivatives. Here h is a sufficiently small increment. The proper choice of h, however, is not easy: A too small value causes a cancellation of significant digits in the difference, whereas a too large value increases the error due to the $\mathrm{O}(h^2)$ term. The latter problem was partially solved by Romberg, whose algorithm makes the $O(h^{2n})$ terms cancel by means of a clever superposition of several finite differences [8], and by Ridder, who combined Romberg’s method with an iterative scheme that yields the smallest possible error of the derivative [9]. Ridder’s method is sufficient for many applications, but certainly not for all, particularly if higher-order derivatives are needed.

An evident remedy is the use of computer arithmetics of more than standard “double” precision. In C++ programs, for instance, this can be accomplished with the `boost::multiprecision` library. Unfortunately, the use of higher-than-double precision arithmetics significantly slows down the program execution.

Numerical differentiation with almost machine precision, even for higher-order derivatives, is possible with “complex integration” methods. These are based on Cauchy’s integral theorem, which relates the path integral along a closed path around a point in the complex plane to derivatives at that point. For derivatives at real-valued locations the method of Lyness and Moler [10] can be recommended, which determines the number of integration nodes automatically while avoiding superfluous invocations of the argument function. A short description is given in Sec. 8. A disadvantage of the method is, however, that it cannot be applied directly to mixed derivatives to functions of more than one variable. This requires either a generalization of the method which involves multi-dimensional integration or the combination of several derivatives through the formalism of directional derivatives (as proposed, for instance, by Deiters and Bell [11, supporting information]).

In this article we want to draw attention to an alternative technique for numerical differentiation that is applicable to pure as well as mixed derivatives of arbitrary order and yields derivatives with almost machine precision, namely the multicomplex step method. This technique is not new, but we feel that it is not as well known as it should be.

2. Theory

Before we come to the discussion of the differentiation methods it is necessary to emphasize that we are, in principle, seeking derivatives of real-valued functions for real-valued arguments. Even if the methods that are described in the following sections require a generalization of the argument domain as well as the co-domain to complex (or even multicomplex) numbers, the functions considered here have the property that they return a real value if their arguments are all real,

$$\begin{gathered} \boldsymbol{x} = (x_1, \ldots, x_N) \\ \boldsymbol{x} \in \mathbb{R}^N \;\Rightarrow\; f(\boldsymbol{x}) \in \mathbb{R}\,. \end{gathered} \tag{2}$$

We call such functions "extended real" in order to distinguish them from general complex functions.

2.1 The Squire–Trapp method

The method of Squire and Trapp [12, 13] is, in principle, a finite-difference method. It is based on the well known definition of the first-order derivative,

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h} \text{ with } x, h \in \mathbb{R}\,. \tag{3}$$

In contrast to the classical finite-element methods, however, the function argument is now incremented along the imaginary axis,

$$\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h\to 0} \Re\left\{\frac{f(x+hi)-f(x)}{hi}\right\} = \lim_{h\to 0} \Im\left\{\frac{f(x+hi)-f(x)}{h}\right\}\,. \tag{4}$$

The $\Re$ and $\Im$ operators return the real or imaginary, respectively, part of the complex function value. Now $\Im\{f(x)\}$ is zero by definition for extended-real functions. The differentiation formula thus becomes

$$\boxed{\frac{\mathrm{d}f(x)}{\mathrm{d}x} = \lim_{h\to 0} \frac{\Im\{f(x+hi)\}}{h}\,.} \tag{5}$$

As no difference needs to be computed anymore, it is possible to use very small increments without any loss of precision, e.g., $h \approx 10^{-100}$.

2.2 Definition of multicomplex numbers

The extension of the Squire–Trapp method to higher-order derivatives requires a generalization of complex numbers to so-called multicomplex numbers. Recent explorations of multicomplex algebra cover some of the relevalant numerical considerations expanded upon here [14, 15]

A complex number can be written as

$$z = r_0 + i r_1\,, \tag{6}$$

where r_0 and r_1 are real numbers and $i \equiv i^{(1)}$ the imaginary unit. The set of complex numbers can be represented by a plane spanned by a real and an imaginary axis.

For bicomplex numbers a third axis is added, which is characterized by its own imaginary unit, $i^{(2)}$. Just as a complex number comprises two real numbers, a bicomplex number b comprises two complex numbers z_0 and z_1

$$\begin{aligned} b &= z_0 + i^{(2)} z_1 \\ &= r_{00} + i^{(1)} r_{01} + i^{(2)} r_{10} + i^{(1)} i^{(2)} r_{11} \,. \end{aligned} \tag{7}$$

Similarly, a tricomplex number comprises two bicomplex numbers, and so on.

The general recursive definition of a multicomplex number of level l is

$$m^{(l)} = \begin{cases} m_0^{(l-1)} + i^{(l)} m_1^{(l-1)} & \text{if } l \geq 2 \\ r_0 + i^{(1)} r_1 & \text{if } l = 1 \end{cases} \tag{8}$$

Thus a multicomplex number of level $l = 1$ corresponds to a complex number. Level $l = 0$ is not allowed. The imaginary units of all levels obey the rule

$$\left(i^{(l)}\right)^2 = -1 \,. \tag{9}$$

Taken together, Eqs. (8) and (9) permit the evaluation of multicomplex expressions and functions by recursively referring them to complex or real arithmetics. This is explained in more detail in Sec. 6.

A multicomplex number of level l can be regarded as a binary tree, where the leaves are the real components. Evidently, there are 2^l real components r_k with $k = 0, \ldots, 2^l - 1$. The binary representation of the subscript k may be regarded as "coordinate" of a real component in the tree, for example,

$$m^{(3)} = \begin{cases} m_0^{(2)} = \begin{cases} m_{00}^{(1)} = \begin{cases} r_0 & 000 \\ r_1 & 001 \end{cases} \\ m_{01}^{(1)} = \begin{cases} r_2 & 010 \\ r_3 & 011 \end{cases} \end{cases} \\ m_1^{(2)} = \begin{cases} m_{10}^{(1)} = \begin{cases} r_4 & 100 \\ r_5 & 101 \end{cases} \\ m_{11}^{(1)} = \begin{cases} r_6 & 110 \\ r_7 & 111 \end{cases} \end{cases} \end{cases} \tag{10}$$

For better clarity, we introduce a "component operator" $\mathfrak{C}_k\{\}$ according to

$$\mathfrak{C}_k\left\{m^{(l)}\right\} = r_k \,, \tag{11}$$

i.e., this operator returns the kth real component of a multicomplex number. The binary representation of k indicates the sequence of real/imaginary forks taken when traversing the tree. Component $\mathfrak{C}_0\left\{m^{(l)}\right\} = r_0$ is found in this tree by always taking the real fork; it is the "most real" component. Conversely, always taking the imaginary fork leads to the "most imaginary" component, $\mathfrak{C}_{2^l-1}\left\{m^{(l)}\right\} = r_{2^l-1}$.

For a multicomplex number of level 1 (i.e., a complex number), $\mathfrak{C}_0\{\}$ corresponds to the $\Re$ operator, and $\mathfrak{C}_1\{\}$ corresponds to the $\Im$ operator.

2.3 Multicomplex differentiation—functions of a scalar

Let $f(x)$ denote a real function of the real variable x. It is assumed that f can be differentiated at least l times with respect to x.

As a generalization of the Squire–Trapp method, we now extend x to a multicomponent number of level l. Consequently, $f(x)$ also becomes multicomplex. We now consider the following Taylor series:

- $l = 1$:

$$\begin{aligned} y &\equiv f(x+i^{(1)}h) \\ &= f_0 + i^{(1)}hf_1 + \frac{1}{2}\left(i^{(1)}\right)^2 h^2 f_2 + \frac{1}{6}\left(i^{(1)}\right)^3 h^3 f_3 + \ldots \\ &= f_0 + i^{(1)}hf_1 - \frac{1}{2}h^2 f_2 - \frac{1}{6}i^{(1)}h^3 f_3 + \ldots \\ &\text{with } f_k \equiv \frac{\mathrm{d}^k f(x)}{\mathrm{d}x^k} \end{aligned} \tag{12}$$

  Collecting the real and imaginary terms then leads to

$$\mathfrak{C}_0\{y\} = f_0 - \frac{h^2}{2}f_2 + \ldots \approx f_0 \tag{13}$$

  and

$$\mathfrak{C}_1\{y\} = hf_1 - \frac{h^3}{6}f_3 + \ldots \approx hf_1\ . \tag{14}$$

  As h can be made arbitrarily small, these equations yield the function value f_0 and the 1st-order derivative f_1 practically exactly. Eq. (14) is equivalent to the Squire–Trapp formula, Eq. (5).

- $l = 2$:

$$\begin{aligned} y \equiv f(x+i^{(1)}h+i^{(2)}h) &= f_0 + \left(i^{(1)}+i^{(2)}\right)hf_1 + \frac{1}{2}\left(i^{(1)}+i^{(2)}\right)^2 h^2 f_2 \\ &+ \frac{1}{6}\left(i^{(1)}+i^{(2)}\right)^3 h^3 f_3 + \ldots \\ &= f_0 + \left(i^{(1)}+i^{(2)}\right)hf_1 + \left(-1+i^{(1)}i^{(2)}\right)h^2 f_2 \quad + \left(-\frac{1}{2}i^{(1)} - \frac{1}{2}i^{(2)}\right)h^3 f_3 + \ldots \end{aligned} \tag{15}$$

  The components of this multicomplex number are (omitting negligible h^2 terms)

$$\begin{aligned} \mathfrak{C}_0\{y\} &\approx f_0 \\ \mathfrak{C}_1\{y\} = \mathfrak{C}_2\{y\} &\approx hf_1 \\ \mathfrak{C}_3\{y\} &\approx h^2 f_2 \end{aligned} \tag{16}$$

  Evidently the evaluation of this multicomplex function yields not only the 2nd-order derivative, but also the 1st-order derivative and the function value.

These results can be generalized for arbitrary levels l: The lth-order derivative of a function is obtained as

$$\begin{aligned} \frac{\mathrm{d}^l f(x)}{\mathrm{d}x^l} &= \lim_{h\to 0} \frac{\mathfrak{C}_{2^l-1}\left\{f(m^{(l)})\right\}}{h^l} \\ \text{with } m^{(l)} &= x + i^{(1)}h + \ldots + i^{(l)}h\ , \end{aligned} \tag{17}$$

where h is a small increment. A proper choice is discussed in Section 3.1. For the practical evaluation the multicomplex argument of f can be set up as follows,

$$\mathfrak{C}_k\left\{m^{(l)}\right\} = \begin{cases} x & k = 0 \\ h & k = 1, 2, \ldots, 2^{l-1} \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

Eq. (17) is the generalization of the Squire–Trapp method for higher-order derivatives. It should be noted that the last, “most imaginary”, component of $f(m^{(l)})$ contains the highest-order derivative. The other derivatives can be extracted from other components of $f(m^{(l)})$, too. The general formula for the calculation of derivatives of a function of a scalar is

$$\boxed{\frac{\mathrm{d}^k f(x)}{\mathrm{d}x^k} = \lim_{h\to 0} \frac{\mathfrak{C}_{2^k-1}\left\{f(m^{(l)})\right\}}{h^k}, \; k = 0,\ldots,l \; .} \tag{19}$$

As with the Squire–Trapp method, the increment h can be made as small as the internal number representation of a computer allows, so that the derivatives can be obtained practically with machine precision.

2.4 Multicomplex differentiation – functions of a vector

A very attractive feature of multicomplex differentiation is its applicability to functions of more than one variable. Thus it is possible to obtain the derivatives of a function of a vector $\boldsymbol{x} \equiv (x_1,\ldots,x_n)$, including mixed derivatives, according to

$$\begin{gathered}\left(\frac{\partial^l f(\boldsymbol{x})}{\partial x_1^{k_1}\ldots\partial x_n^{k_n}}\right) = \lim_{h\to 0} \frac{\mathfrak{C}_{2^l-1}\left\{f\left(\boldsymbol{m}^{(l)}\right)\right\}}{h^l} \\ \text{with } \boldsymbol{m}^{(l)} = \begin{pmatrix} x_1 + h\sum_{j=1}^{k_1} i^{(j)} \\ \vdots \\ x_n + h\sum_{j=l-k_n+1}^{l} i^{(j)} \end{pmatrix} \text{ and } \sum_{j=1}^{n} k_j = l \; .\end{gathered} \tag{20}$$

Each component of the multicomplex argument vector $\boldsymbol{m}^{(l)}$ receives as many increment terms as the order of the differentiation, k_i, prescribes. To obtain, for example, the mixed third derivative $\left(\partial^3 f(\boldsymbol{x})/(\partial x_1^2\,\partial x_2)\right)$ it is necessary to evaluate

$$\begin{gathered}\left(\frac{\partial^3 f(\boldsymbol{x})}{\partial x_1^2\,\partial x_2}\right) = \lim_{h\to 0} \frac{\mathfrak{C}_7\left\{f\left(\boldsymbol{m}^{(3)}\right)\right\}}{h^3} \\ \text{with } \boldsymbol{m}^{(3)} = \begin{pmatrix} x_1 + i^{(1)}h + i^{(2)}h \\ x_2 + i^{(3)}h \end{pmatrix}\end{gathered} \tag{21}$$

Whether an $i^{(j)}h$ increment is added to x_1 or to x_2 does not matter, as long as x_1 gets two increments and x_2 gets one.

Just as with the differentiation of functions of scalars, the multicomplex function result contains all lower-order derivatives, too. Their extraction depends on the association of the imaginary units $i^{(j)}$ with the argument components x_i. For the association scheme employed in Eq. (20), the formula for the determination of a derivative having the differentiation orders $k_1' \leq k_1, \ldots k_n' \leq k_n$ is

$$\boxed{\begin{gathered}\left(\frac{\partial^K f(\boldsymbol{x})}{\partial x_1^{k_1'}\ldots\partial x_n^{k_n'}}\right) = \lim_{h\to 0} \frac{\mathfrak{C}_J\left\{f\left(\boldsymbol{m}^{(l)}\right)\right\}}{h^K} \\ \text{with } K \equiv \sum_{j=1}^{n} k_j' \; .\end{gathered}} \tag{22}$$

The index J is given by

$$J \equiv \sum_{j=1}^{n} \left((2^{k'_j} - 1) 2^{b_j} \right)$$
$$\text{with } b_j = \begin{cases} 0 & \text{j} = 0 \\ \sum_{j_1=1}^{j-1} k_{j_1} & j > 0 \end{cases} , \tag{23}$$

where the b_j are the accumulated orders of differentiation that were used to set up $\boldsymbol{m}^{(l)}$. This expression for J may look complicated, but its implementation in a computer program is not, as shown in Section 3.3. For the example given above, Eq. (21), the b exponents are $b_1 = 0$ and $b_2 = 2$. Then the mixed 2nd-order derivative ($k'_1 = k'_2 = 1$) is

$$\left(\frac{\partial^2 f(\boldsymbol{x})}{\partial x_1 \, \partial x_2} \right) = \lim_{h \to 0} \frac{\mathfrak{C}_5 \left\{ f \left(\boldsymbol{m}^{(3)} \right) \right\}}{h^2} , \tag{24}$$

and the pure 2nd-order derivative with respect to x_1 ($k'_1 = 2$, $k'_2 = 0$) is

$$\left(\frac{\partial^2 f(\boldsymbol{x})}{\partial x_1^2} \right) = \lim_{h \to 0} \frac{\mathfrak{C}_3 \left\{ f \left(\boldsymbol{m}^{(3)} \right) \right\}}{h^2} , \tag{25}$$

These two (and more) derivatives are obtained without additional computational costs when $\left(\partial^3 f(x) / (\partial x_1^2 \, \partial x_2) \right)$ is computed.

3. Application

3.1 The differentiation increment

It is a pleasant feature of multicomplex differentiation that there is no need to provide a problem-specific control parameter. It is necessary to choose a differentiation increment h, but this should simply be made as small as possible. This is in contrast to real finite-difference methods, where the determination of an optimal step width that balances the round-off errors and the termination errors is a nontrivial task. Complex-integration methods, on the other hand, require an integration radius, and an improper choice either spoils the accuracy or slows the calculation down[16].

However, the proposition to make h as small as possible needs some further explanations. Higham proposes $h = 10^{-100}$ for the Squire–Trapp method, but this value cannot be adopted for multicomplex differentiation without further consideration. The formulas for lth-order derivatives, Eqs. (17) and (20), contain the term h^l. Now standard double-precision arithmetics (IEEE 754-2019 [17]) imposes a lower boundary for this power, approximately $h^l > 10^{-308}$. It is therefore advisable to let h depend on l. We found that $h \approx 2^{-664/l}$ works well[1].

There is, however, a problem of which the users of multicomplex differentiation should be aware: The differentiation formulas Eqs. (17) and (20) rely on the fact that $O(h^2)$ terms from the underlying Taylor expansions can be neglected. This leads to the approximate restriction $h^2 < \varepsilon$, where $\varepsilon \approx 10^{-15}$ is the relative precision of double-precision arithmetics. This limits the order of differentiation to about 12 (when using the formula for h given above).

[1]C++ code: `double h = ldexp(1.0, -664 / l);`

3.2 Numerical effort

Another consideration is that a multicomplex number of level l comprises 2^l real components (e.g., 1024 components for a 10th-order derivative!). One can therefore expect the storage requirements as well as the computing time to increase exponentially with the order of differentiation.

In contrast to this, the numerical effort of differentiation by complex integration increases approximately linearly with l, so that such methods would probably be preferable for higher-order differentiations.

Fortunately, most problems in fluid thermodynamics require 1st- and 2nd-order derivatives only, e.g., the computation of entropy, enthalpy, chemical potential or compressibility from a fundamental equation (Helmholtz energy equation). The calculation of critical curves of mixtures (including stability analysis) requires up to 4th-order derivatives. At present, 5th- and higher-order derivatives appear in calculations of tricritical states or higher-order critical states of similar complexity only, which are needed for the construction of global phase diagrams [18–20]. All these applications are therefore within the useful range of multicomplex differentiation.

3.3 Implementation hints

Multicomplex arithmetics is not (yet) routinely included in most modern computer languages. We therefore make multicomplex libraries for C++ and Python available [21]. Furthermore, we describe basic multicomplex operations as well as multicomplex versions of some frequently used functions in Sec. 6. Section 7 offers some commented C++ code for multicomplex differentiation.

In this section we demonstrate the application of multicomplex differentiation to some thermodynamic problems.

In our C++ implementation, multicomplex numbers are instances of a `multicomplex` class. The $\mathfrak{C}\{\}$ operator, which sets or retrieves the real components of a multicomplex number, is implemented as a member function of that class:

```
double x, y;
multicomplex m;
...
m.r_idx(0, x);  // set real component with index 0 to x
y = m.r_idx(3); // copy real component 3 to y
```

For our application examples we define the dimensionless residual Helmholtz energy function of a pure fluid as a function of the molar density ρ and the temperature T; for simplicity we use the model of van der Waals here,

$$\alpha^{\mathrm{r}} \equiv \frac{A^{\mathrm{r}}_{\mathrm{m}}}{RT} = -\ln(1-b\rho) - \frac{a\rho}{RT} \tag{26}$$

where a and b are substance-specific parameters; of course more realistic (and more complicated) Helmholtz energy functions can be used instead. We implement the van der Waals Helmholtz energy function in C++ as

```
double alphar(double rho, double T, double a, double b) {
  double xi = b * rho; // reduced density
  return -log(1.0 - xi) - a * xi / (b * R * T);
}
```

In order to create a multicomplex version of this function, we merely have to add a template declaration:

```
template <typename X>
X alphar(X rho, X T, double a, double b) {
  X xi = b * rho; // reduced density
  return -log(1.0 - xi) - a * xi / (b * R * T);
}
```

3.3.1 The pressure of a pure fluid

The pressure is obtained as

$$p = -\left(\frac{\partial A_{\mathrm{m}}}{\partial V_{\mathrm{m}}}\right)_T = RT\rho + RT\rho^2\left(\frac{\partial \alpha^{\mathrm{r}}}{\partial \rho}\right)_T . \tag{27}$$

The following C++ code snippet calculates the pressure by multicomplex differentiation:

```
multicomplex rhom, y;
rhom.init(1);                   // set level to 1 (1st-order diff.)
y.init(1);
rhom.r_idx(0, rho);             // set argument value
rhom.r_idx(1, h);               // set increment
y = alphar(rhom, T, a, b);
p = R * T * rho * (1.0 + rho * y.r_idx(1) / h);
```

3.3.2 Virial coefficients of a pure fluid

The (density-dependent) virial series is

$$Z \equiv \frac{p}{\rho RT} = 1 + B_2\rho + B_3\rho^2 + \ldots \tag{28}$$

The virial coefficients can be interpreted as coefficients of a Taylor series of $\alpha^{\mathrm{r}}(\rho)$,

$$B_{l+1} = \frac{1}{(l-1)!}\left(\frac{\partial^l \alpha^{\mathrm{r}}(\rho,T)}{\partial \rho^l}\right),\ l = 1,2,\ldots \tag{29}$$

at the density $\rho = 0$. The corresponding C++ code is

```
unsigned int j, k, l, jfac;
multicomplex rhom, y;
rhom.init(l);                   // get (l+1)th virial coefficient
y.init(l);
rhom.r_idx(0, 0.0);
for (j = 1, k = 1; j <= l; j++) {
  rhom.r_idx(k, h);             // increment components 2^j
  k <<= 1;                      // multiplication by 2
}
y = alphar(rhom, T);
for (j = 1, jfac = 1, k = 1, hl = 1.0; j <= l; j++) {
  hl *= h;                      // h^j
  k <<= 1;
  B[j] = y.r_idx(k - 1) / ( jfac * hl);  // results
}
jfac *= j;
```

It should be noted that a single evaluation of the multicomplex Helmholtz energy function generates all virial coefficients up to B_{l+1} at once.

3.3.3 The isochoric pressure coefficient

This coefficient is the temperature derivative of the pressure, and therefore a mixed derivative of $\alpha^{\mathrm{r}}(\rho,T)$,

$$\frac{\beta_V}{\rho R} = \frac{1}{\rho R}\left(\frac{\partial p}{\partial T}\right)_{\rho} = 1 + \rho\left(\frac{\partial \alpha^{\mathrm{r}}}{\partial \rho}\right) + \rho T\left(\frac{\partial^2 \alpha^{\mathrm{r}}}{\partial \rho\,\partial T}\right) . \tag{30}$$

The corresponding C++ code is

```
multicomplex rhom, Tm, y;
rhom.init(2);                  // need 2nd-order derivatives
Tm.init(2);
y.init(2);
rhom.r_idx(0, rho);
rhom.r_idx(1, h);
Tm.r_idx(0, T);
Tm.r_idx(2, h);
y = alphar(rhom, Tm, a, b);
betaV = 1.0 + rho * (y.r_idx(1) / h + T * y_r_idx(3) / (h * h));
```

In this case the highest orders of differentiation with respect to ρ and T are both 1 ($k_1 = k_2 = 1, l = 2$). The b_j coefficients (Eq. (23)) are 0 and 1, and therefore $(\partial\alpha^{\mathrm{r}}/\partial\rho)$ (with $k_1' = 1$, $k_2' = 0$) is found at the index $J = 1$.

It should be noted that again merely one multicomplex function call is needed to obtain both partial derivatives.

3.3.4 Functions of a vector—general implementation

A general recipe for setting up the multicomplex arguments for a differentiation of a function an n-dimensional argument, $f(\boldsymbol{x})$, with the orders $k_1, k_2, \ldots, k_n$ is:

```
int i, j;
vector<unsigned int> k(n);
unsigned int m, l;
vector<multicomplex> xm(n);
for (j = 0, l = 0; j < n; j++) l += k[j]; // total order of differentiation
for (j = 0, m = 0x01; j < n; j++) {
  xm[j].init(l);
  (xm[j]).r_idx(0, x[j]);      // set function arguments
  for (i = 1; i <= k[j]; i++) {
    xm[j].r_idx(m, h);         // set increments
    m <<= 1;                   // bit shift = multiplication with 2
  }
}
```

This code snippet follows the C/C++ array index convention—i.e., arrays start with index 0—and assumes that a convenient array class has been defined. Here the class `std::vector` is used, but there are several alternative classes.

The next step is the evaluation of the multicomplex function:

```
multicomplex ym;
ym.init(l);
ym = f(xm);
```

The highest-order derivative is always related to the last, "most imaginary" component of the function value.

```
derivative = ym.r_idx(exp2i(l) - 1) * pow(h, -l);
```

where the function for two's exponentiation can be given as a bitshift operation

```
int exp2i(int j){
    return 1 << j;
}
```

If we also wish to extract the derivative with the orders $k'_1, k'_2, \ldots, k'_n$ (with $k' \leq k$), it is first necessary to build the $\boldsymbol{b}$ vector,

```
vector<int> b(n);
for (j = 1, b[0] = 0; j < n; j++) b[j] = b[j - 1] + k[j - 1];
```

and then execute the following loops for each desired derivative,

```
for (j = 0, lp = 0; j < n; j++) lp += kp[j]; // order of this derivative, l'
for (j = 0, m = 0x00; j < n, j++) m += ((0x01 << kp[j]) - 0x01) << b[j];
derivative = ym.r_idx(m) * pow(h, -lp);
```

4. Results

In this section we time some calculations with the multicomplex approach. Benchmark results in C++ from the library are included in Table 1 for some common mathematical functions. Aside from the `ln(x)` function (and the `pow` function which implicitly calls the `ln(x)` function), for which a few microseconds are required to evaluate the first derivative, the first derivatives of all other mathematical functions are evaluated in less than a microsecond. To put that in perspective, there is on the order of 1 μs overhead to make a call from Python to C++, so the computational penalty is not too severe. Other approaches, most notably automatic differentiation approaches, have much lower computational overhead, but introduce some new challenges from an implementation standpoint. The benchmark script is available in the source code of `multicomplex`.

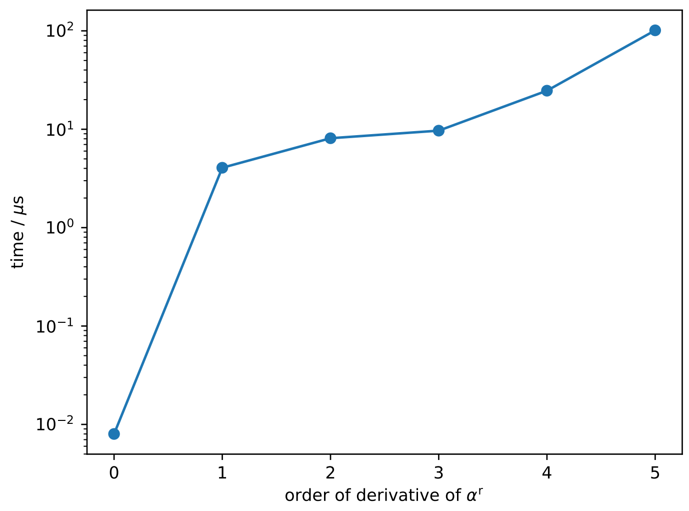

Fig. 1. Timing of the density derivatives $\left(\frac{\partial^n(\alpha^r)}{\partial\rho^n}\right)_T$ of the van der Waals equation of state in C++.

Table 1. Timing results for some common mathematical expressions. The test name matches the same field in the C++ code of the `bench.cxx` file of `multicomplex`. Tests were run on an Intel i7-10810U CPU running windows 10 with the benchmarking tools of the Catch2 package, and the XML output was processed to generate the table.

test	function / ns	first derivative / ns
`time x^3`	1.0	458.0
`time x^3, but with pow(x,int)`	42.0	671.0
`time x^3, but with pow(x,double)`	40.0	6676.0
`time 1/x`	1.0	1066.0
`time sin(x)`	13.0	452.0
`time cosh(x)`	16.0	555.0
`time exp(x)`	9.0	448.0
`time ln(x)`	7.0	5878.0
`time cos(x)*sin(x)`	20.0	659.0

To come back to the van der Waals equation of state, we provide timing of the density derivatives of the equation of state in Fig. 1, in C++, as a function of the number of derivatives desired. We should caution that this is a somewhat unfair comparison because the evaluation of the EOS itself requires only a few computational cycles as it invokes a logarithm but otherwise only the negation, subtraction, multiplication, and division operators, each of which can be very efficiently carried out in double precision. The jump from the function evaluated in double precision to first derivative is rather steep, but after that, higher derivatives do not introduce much additional computational penalty.

5. Conclusion

Numerical differentiation with the multicomplex finite-step method is a very versatile and easy-to-use tool for calculating derivatives of functions with nearly machine precision. In contrast to the Squire–Trapp method, of which the multicomplex step method is an extension, it is applicable to higher-order derivatives. In contrast to methods based on complex integration, the multicomplex step method can obtain mixed derivatives without introducing computational complications like integration in more than one dimension. Moreover, the multicomplex method can obtain several derivatives of a function at the same time without additional computing effort.

Another pleasant feature of the new method is that there is no control parameter that needs to be determined or optimized, like the differentiation increment of real finite-step methods (e.g., the methods of Romberg or Ridder) or the integration radius of the complex-integration methods. The multicomplex step method does use an increment, but its value can be chosen within several orders of magnitude; choosing the increment size is uncritical and can be safely left to the algorithm.

There are, however, some limitations: The attainable size of the increment on a computer using double-precision arithmetics as well as CPU time considerations restrict the useful range of the order of differentiation to approximately 12. But this is more than enough for most thermodynamical applications—calculation of phase equilibria or caloric properties from fundamental equations (Helmholtz energy as a function of density, temperature, and composition) or of critical states.

A practical disadvantage is, at present, that the contemporary computer languages cannot handle multicomplex numbers. We therefore describe the implementation of multicomplex calculations—arithmetic operations as well as some important functions—in Sec. 6. Ready-to-use program libraries for C++ and Python can be found in Ref. [21]; the C++ library can also be invoked from Fortran programs. Furthermore, we provide Python sample code in Section 9.

6. Appendix A: Multicomplex arithmetics and functions

As multicomplex arithmetics is not widely known, yet, and not included in standard compiler libraries, we present here some rules for the manipulation of multicomplex numbers as well as methods for the evaluation of some frequently needed functions.

As will be explained below, some well-behaved real functions have got rather problematic complex or multicomplex counterparts. But even such functions can be evaluated reliably for real-dominant arguments. Fortunately, this is exactly the case for multicomplex differentiation.

The differentiation recipes Eqs. (19), (20), and (22) work reliably if these conditions are fulfilled:

1. If the multicomplex argument $m^{(l)}$ is effectively real (i.e., all $\mathfrak{C}_{k\neq 0}\left\{m^{(l)}\right\} = 0$), then $f(m^{(l)})$ must be real, too.
2. If the multicomplex argument is real-dominant (i.e. all $\mathfrak{C}_{k!=0}\left\{m^{(l)}\right\} = \mathrm{O}(h)$), then $f(m^{(l)})$ must be real-dominant, too.
3. The differentiation increment h must be small.

Python and C++ code for the implementation of multicomplex arithmetics can be found here [21]. The C++ library can also be invoked from Fortran programs.

6.1 Arithmetic operations

Let $a^{(l)}$ and $b(l)$ denote two multicomplex numbers of level l. Then their sum is obtained as

$$c^{(l)} = a^{(l)} + b^{(l)} : \quad \begin{aligned} c_0^{(l-1)} &= a_0^{(l-1)} + b_0^{(l-1)} \\ c_1^{(l-1)} &= a_1^{(l-1)} + b_1^{(l-1)} \ , \end{aligned} \tag{31}$$

their difference as

$$c^{(l)} = a^{(l)} - b^{(l)} : \quad \begin{aligned} c_0^{(l-1)} &= a_0^{(l-1)} - b_0^{(l-1)} \\ c_1^{(l-1)} &= a_1^{(l-1)} - b_1^{(l-1)} \ , \end{aligned} \tag{32}$$

their product as

$$c^{(l)} = a^{(l)} b^{(l)} : \quad \begin{aligned} c_0^{(l-1)} &= a_0^{(l-1)} b_0^{(l-1)} - a_1^{(l-1)} b_1^{(l-1)} \\ c_1^{(l-1)} &= a_0^{(l-1)} b_1^{(l-1)} + a_1^{(l-1)} b_0^{(l-1)} \ , \end{aligned} \tag{33}$$

and their quotient as

$$c^{(l)} = \frac{a^{(l)}}{b^{(l)}} : \quad \begin{aligned} c_0^{(l-1)} &= \frac{a_0^{(l-1)} b_0^{(l-1)} + a_1^{(l-1)} b_1^{(l-1)}}{\left(b_0^{(l-1)}\right)^2 + \left(b_1^{(l-1)}\right)^2} \\ c_1^{(l-1)} &= \frac{a_1^{(l-1)} b_0^{(l-1)} - a_0^{(l-1)} b_1^{(l-1)}}{\left(b_0^{(l-1)}\right)^2 + \left(b_1^{(l-1)}\right)^2} \ . \end{aligned} \tag{34}$$

Unless the divisor is zero in the last case, all these operations can be applied to arbitrary multicomplex numbers and are therefore safe to use.

Addition of and multiplication by real numbers are described here:

$$c^{(l)} = a^{(l)} + x : \quad \begin{aligned} c_0^{(l-1)} &= a_0^{(l-1)} + x \\ c_1^{(l-1)} &= a_1^{(l-1)} \end{aligned} \tag{35}$$

$$c^{(l)} = a^{(l)} x : \quad \begin{aligned} c_0^{(l-1)} &= a_0^{(l-1)} x \\ c_1^{(l-1)} &= a_1^{(l-1)} x \end{aligned} \tag{36}$$

For level $l = 1$ the well-known rules for complex arithmetics can be used.

Finally it is useful to define a "promotion rule": If an arithmetic operation or a value assignment is performed with multicomplex numbers of different levels, the number with the lower level gets promoted,

$$a^{(l)} = a^{(l-1)} + i^{(l)} \cdot 0 \,. \tag{37}$$

6.2 Comparison operations

Two multicomplex numbers are equal if all their real components are equal, and unequal otherwise. The recursive formulation of this rule is

$$\begin{aligned} a^{(l)} = b^{(l)} : \quad & a_0^{(l-1)} = b_0^{(l-1)} \\ & \qquad \wedge \\ & a_1^{(l-1)} = b_1^{(l-1)} \\ a^{(l)} \neq b^{(l)} : \quad & a_0^{(l-1)} \neq b_0^{(l-1)} \\ & \qquad \vee \\ & a_1^{(l-1)} \neq b_1^{(l-1)} \end{aligned} \tag{38}$$

As already for complex numbers, the comparison operators $<$ and $>$ cannot be defined. It is, however, possible to define the norm of a multicomplex number, which is the square root of the sum of its real components,

$$\left\| a^{(l)} \right\| \equiv \sqrt{\left(a_0^{(l-1)}\right)^2 + \left(a_1^{(l-1)}\right)^2} \,, \tag{39}$$

and therefore a real number. This norm can be used, for instance, to terminate iteration loops.

6.3 Exponential and trigonometric functions

These functions can be implemented as follows [22]:

$$y^{(l)} = \sin(x^{(l)}) : \quad \begin{aligned} y_0^{(l-1)} &= \sin(x_0^{(l-1)}) \cosh(x_1^{(l-1)}) \\ y_1^{(l-1)} &= \cos(x_0^{(l-1)}) \sinh(x_1^{(l-1)}) \end{aligned} \tag{40}$$

$$y^{(l)} = \cos(x^{(l)}) : \quad \begin{aligned} y_0^{(l-1)} &= \cos(x_0^{(l-1)}) \cosh(x_1^{(l-1)}) \\ y_1^{(l-1)} &= -\sin(x_0^{(l-1)}) \sinh(x_1^{(l-1)}) \end{aligned} \tag{41}$$

$$y^{(l)} = \sinh(x^{(l)}): \quad \begin{aligned} y_0^{(l-1)} &= \sinh(x_0^{(l-1)})\cos(x_1^{(l-1)}) \\ y_1^{(l-1)} &= \cosh(x_0^{(l-1)})\sin(x_1^{(l-1)}) \end{aligned} \tag{42}$$

$$c^{(l)} = \cosh(a^{(l)}): \quad \begin{aligned} y_0^{(l-1)} &= \cosh(x_0^{(l-1)})\cos(x_1^{(l-1)}) \\ y_1^{(l-1)} &= \sinh(x_0^{(l-1)})\sin(x_1^{(l-1)}) \end{aligned} \tag{43}$$

$$y^{(l)} = \exp(a^{(l)}): \quad \begin{aligned} y_0^{(l-1)} &= \exp(x_0^{(l-1)})\cos(x_1^{(l-1)}) \\ y_1^{(l-1)} &= \exp(x_0^{(l-1)})\sin(x_1^{(l-1)}) \end{aligned} \tag{44}$$

The disadvantage of this exp() implementation is that a large negative argument, e.g., e^{-1000}, may cause an overflow error in the hyperbolic functions sinh and cosh called by sin and cos. Instead, it is better to implement the exponential function as

$$y^{(l)} = \exp(x^{(l)}) = \exp(\mathfrak{C}_0\left\{a^{(l)}\right\})\left[\cosh\left(\frac{a^{(l)}}{\mathfrak{C}_0\left\{a^{(l)}\right\}}\right) + \sinh\left(\frac{a^{(l)}}{\mathfrak{C}_0\left\{a^{(l)}\right\}}\right)\right] \tag{45}$$

if $|\mathfrak{C}_0\left\{a^{(l)}\right\}| > 1$.

For $l = 1$, the well-known complex functions can be used.

6.4 The natural logarithm function

The natural logarithm of a complex number $z = re^{i\phi}$ is given by

$$\ln z = \ln r + i(\phi + 2\pi k), k = \dots, -2, -1, 0, +1, +2, \dots\ . \tag{46}$$

This is evidently a multivalued relation. The complex log() function of the C/C++ standard library returns the principal value of $\ln(x)$, namely the value for $k = 0$.

The test relation

$$\exp(\ln x) = e^{\ln r} e^{i\phi + 2\pi k} = r(\cos(\phi + 2\pi k) + i\sin(\phi + 2\pi k)) = x\,, \tag{47}$$

returns the argument, because the sine and cosine functions are periodic with a period length of 2π. Hence the relation is fulfilled for any value of $k \in \mathbb{Z}$, the set of integers. The calculation of a square root according to

$$\sqrt{z} = \exp\left(\frac{1}{2}\ln x\right) = \sqrt{r}\left(\cos\left(\frac{\phi}{2} + \pi k\right) + i\sin\left(\frac{\phi}{2} + \pi k\right)\right) \tag{48}$$

is not guaranteed to work, because—depending on the value of k—the sine and cosine terms will change their signs.

The ambiguity of evaluations of the ln() function is even aggravated in multicomplex algebra. A general implementation of this function in multicomplex algebra is therefore difficult, perhaps even impossible.

Fortunately, the multicomplex numbers appearing in the course of numerical differentiation are of the real-dominant type. For such numbers an efficient and reliable iterative scheme can be given. We set $y = \ln x$ and apply Halley's method,

$$y \leftarrow y - \frac{f(y)}{f'(y) - \frac{f(y)f''(y)}{2f'(y)}}\ . \tag{49}$$

Setting $f(y) \equiv e^y - x$ gives, after some transformations,

$$y \leftarrow y - 2\,\frac{1-xe^{-y}}{1+xe^{-y}}\ . \tag{50}$$

This iteration can be carried out with multicomplex numbers. It does not require problematic steps. Halley's method has a convergence order of 3, which means that, in the vicinity of the solution, the number of valid digits triples with each step. If the iteration is started at $y_0 = \ln(\mathfrak{C}_0\{x\})$, convergence is usually achieved after 1–2 steps.

6.5 $\ln(1+x)$

This function can be conveniently evaluated by recursion,

$$\ln\left(1+x^{(l)}\right) = \begin{cases} \frac{\ln(u^{(l)})x^{(l)}}{u^{(l)}-1} & u^{(l)} \neq 1 \\ x^{(l)} & u^{(l)} = 1 \end{cases} \tag{51}$$
$$\text{with } u^{(l)} \equiv x^{(l)} + 1$$

This formulation reduces rounding errors if the argument is very small.

6.6 The arctangent function

Like the logarithm function, the arctangent function is safe for real-dominant multicomplex arguments only. It can be calculated iteratively by means of Newton's method,

$$y \leftarrow y - \frac{f(y)}{f'(y)}\ . \tag{52}$$

Substituting

$$f(y) = \tan y \qquad f'(y) = \frac{1}{(\cos y)^2} \tag{53}$$

yields the iteration formula

$$y \leftarrow y - (\sin y \cos y - x(\cos y)^2)\ . \tag{54}$$

This algorithm can also be used for multicomplex x and y. Convergence is rapid for $|\mathfrak{C}_0\{y\}| \leq 1$. For arguments outside this range, the relation

$$\arctan y = \begin{cases} +\frac{\pi}{2} - \arctan\left(\frac{1}{y}\right) & y[0] > 1 \\ -\frac{\pi}{2} - \arctan\left(\frac{1}{y}\right) & y[0] < -1 \end{cases} \tag{55}$$

should be used. For the purpose of numerical differentiation, the iteration should be started at $y = \mathfrak{C}_0\{x\}$.

6.7 The root function

Iteration can also be used to obtain roots or multicomplex numbers, $y = \sqrt[n]{c}$. Substituting $f(y) = y^n - x$ into Newton's scheme yields the iteration formula

$$y \leftarrow \frac{n-1}{n}y + \frac{x}{ny^{n-1}} \tag{56}$$

for the generalized root function. In particular, the square root is obtained from the well-known iteration formula

$$y \leftarrow \frac{1}{2}\left(y + \frac{x}{y}\right)\ . \tag{57}$$

6.8 The power function

Integer powers, x^n, should be handled by multiplication. Arbitrary real powers can be obtained from

$$\mathrm{pow}(x,r) = x^r = (\mathfrak{C}_0\left\{x\right\})^r \exp\left(r \ln\left(\frac{x}{\mathfrak{C}_0\left\{x\right\}}\right)\right) . \tag{58}$$

7. Appendix B: Commented C++ subroutines for multicomplex differentiation

The following samples demonstrate the layout of some multicomplex differentiation subroutines. We do not list all required C++ headers (`*.h` files) here, as these are rather user-specific.

7.1 The increment function

The following function computes an increment for an lth order multicomplex differentiation.

```
inline double increment(unsigned int l) {
  return ldexp(1.0, -664 / static_cast<int>(l));
}
```

7.2 Differentiation of a function of a scalar

The next function computes the lth-order derivative of a multicomplex (extended-real!) function $f(x)$ at the location x.

```
double diff_mcx(function<multicomplex(const multicomplex&)> f,
unsigned int l, double x) {
  unsigned int i, j;
  double hl, h;
  multicomplex xh, yh;
  xh.init(l);        // initialize (provide memory for 2^l components)
  yh.init(l);
  h = increment(l); // estimate a useful increment size
  xh = 0.0;          // set all components of xh to zero
  xh.r_idx(0, x);    // set the most real component to x
  for (i = 1, j = 1, hl = 1.0; i <= l; i++) {
    xh.r_idx(j, h); // set components 1, 2, 4, ... to h
    hl *= h;         // h^l
    j *= 2;
  }
  yh = f(xh);        // evaluate the multicomplex function
  return yh.r_idx(j - 1) / hl;
}
```

The following version of the subroutine returns more than one derivative. The derivatives are specified by means of an “order vector”. If for example, the first and the second-order derivatives are needed, the order vector is a two-component vector defined as $\vec{o} = (1,2)$. We assume that an appropriate vector class has been defined.

```
vector<double> diff_mcx(function<multicomplex(const multicomplex&)> f,
const vector<unsigned int>& o, double x) {
  int i,
    m = o.size();    // number of desired derivatives
  unsigned int j, l;
  for (i = 1, l = o[0]; i < m; i++) l = max(l, o[i]);
                     // max. order of differentiation
```

```
                    // ensure m > 0, l > 0 !
  double h;
  vector<double> d(m), hl(l + 1);
  multicomplex xh, yh;
  xh.init(l);
  yh.init(l);
  h = increment(l);
  xh = 0.0;
  xh.r_idx(0, x);
  hl[0] = 1.0;
  for (i = 1, j = 1; i <= l; i++) {
    xh.r_idx(j, h);
    j *= 2;
    hl[i] = hl[i - 1] * h;
  }
  yh = f(xh);
  for (i = 0; i < m; i++) d[i] = yh.r_idx((0x01 << o[i]) - 1) / hl[o[i]];
  return d;
}
```

The result is a vector containing the derivatives as specified by the order vector.

7.3 Differentiation of a function of a vector

The following example computes a derivative of a multicomplex function $f(\boldsymbol{x})$, where $\boldsymbol{x}$ is an n-dimensional vector. The derivative is specified by means of the "order vector" o, which contains the order of differentiation for each dimension. For example, the second-order mixed derivative of a function of a two-dimensional vector, $\left(\partial^2 f(\boldsymbol{x})/\partial x_0\,\partial x_1\right)$, is specified with $\boldsymbol{o} = (1,1)$.

```
double diff_mcx(function<multicomplex(const Vector<multicomplex>&)> f,
const Vector<unsigned int>& o, const Vector<double>& x) {
  unsigned int m,
    l = o.sum();                    // total order of differentiation, l > 0!
  int i, k,
    n = x.size();
  double hl, h;
  multicomplex yh;
  vector<multicomplex> xh(n); // multicomplex argument vector
  yh.init(l);
  h = increment(l);
  for (k = 0, m = 0x01, hl = 1.0; k < n; k++) { // loop over dimensions
    xh[k].init(l);                 // initialize vector element
    xh[k] = 0.0;
    (xh[k]).r_idx(0, x[k]);        // set the argument value
    for (i = 1; i <= o[k]; i++) {
      (xh[k]).r_idx(m, h);         // set as many components to h as
      m <<= 1;                // specified by o[k]
      hl *= h;
    }
  }
  yh = f(xh);                          // evaluate the function
  return yh.r_idx(m - 1) / hl;         // use the "most imaginary" component
}
```

8. Appendix C: Differentiation by complex integration

8.1 Function of one variable

Complex integration methods are based on Cauchy's integral theorem, which relates the value of a function $f(z)$ or any of its derivatives at the location z_0 to a path integral,

$$\frac{\mathrm{d}^n f(z_0)}{\mathrm{d}z^n} = \frac{n!}{2\pi} \oint_C \frac{f(z+z_0)}{z^n} \mathrm{d}z \,, \tag{59}$$

where C denotes a closed path in the complex plane that encloses z_0. The simplest choice is a circle with the radius r centered at $z = z_0$. Then Cauchy's formula can be written as

$$\frac{\mathrm{d}^n f(z_0)}{\mathrm{d}z^n} = \frac{n!}{2\pi} \int_0^{2\pi} \frac{f(z+z_0)}{z^n} \mathrm{d}\phi \quad \text{with} \quad z \equiv re^{i\phi} \,. \tag{60}$$

The integration can accomplished numerically by means of trapezoidal integration . This is usually known as a rather inferior method, but for periodic functions it is very efficient: With increasing number of evaluation nodes N, the error of the integral decreases faster than any power of $2\pi/N$.

The advantage of using an *integral* formula to obtain derivatives is that numerical integration—quite in contrast to numerical differentiation by means of finite-difference formulas—does not suffer from cancellation errors. If the radius of the integration path is chosen large enough (so large that $f(z+z_0)$ shows a significant variation), derivatives can be computed practically with machine precision.

An efficient application of Cauchy's theorem to extended-real functions is the method of Lyness and Moler [10]. It constructs a series of approximations y_m for the nth-order derivative,

$$\begin{aligned} \frac{\mathrm{d}^n f(x_0)}{\mathrm{d}x^n} &= \lim_{m\to\infty} y_m \\ y_m &= y_{m-1} + \frac{n!}{r^n} M_m \left[-f(x_0) + \frac{1}{h} \sum_{k=0}^{mn-1} \Re\left\{ f\left(x_0 + re^{2\pi hki} \right) \right\} \right], \; m = 1, 2, \ldots \\ &\text{with } y_0 = 0 \text{ and } h = \frac{1}{mn} \,. \end{aligned} \tag{61}$$

The M_m are so-called Möbius numbers, which are defined as

$$\begin{aligned} M_m &= \begin{cases} +1 & \text{if } m = 1 \text{ or if } m \text{ is a product of different prime numbers} \\ -1 & \text{if } m \text{ is a prime number} \\ 0 & \text{otherwise} \end{cases} \\ &= \{+1, -1, -1, 0, -1, +1, -1, 0, 0, +1, -1, 0, -1, +1, +1, 0, -1, 0, -1, 0, \ldots\} \,. \end{aligned} \tag{62}$$

Convergence is usually quite rapid; the calculation can safely be terminated when the difference of two successive approximations falls below a predefined threshold.

8.2 Functions of more than one variable

A straightforward extension of Eqs. (60) or (61) to functions of two or more arguments is possible, but complicated and computationally expensive, as it requires integrations in more than one dimension. This makes it difficult to compute mixed derivatives. Fortunately there is a workaround by means of the formalism of directional derivatives.

The directional derivative of a function of an N-dimensional vector argument, $f(\boldsymbol{x})$ is defined by [23]

$$\frac{\mathrm{d}f(\boldsymbol{x})}{\mathrm{d}\boldsymbol{u}} \equiv \lim_{\xi\to 0}\frac{1}{\xi}\left(f(\boldsymbol{x}+\xi\boldsymbol{u})-f(\boldsymbol{x})\right) . \tag{63}$$

Here $\boldsymbol{u}$ is a direction vector, which can have an arbitrary (but not zero) length.

The total differential of f is

$$\mathrm{d}f = \sum_{i=1}^{N}\left(\frac{\partial f}{\partial x_i}\right)_{x_{j\neq i}}\mathrm{d}x_i . \tag{64}$$

Setting

$$\mathrm{d}x = \boldsymbol{u}\,\mathrm{d}\xi \tag{65}$$

then leads to

$$\mathrm{d}f = \sum_{i=1}^{N}u_i\left(\frac{\partial f}{\partial x_i}\right)_{x_{j\neq i}}\mathrm{d}\xi \tag{66}$$

and, after division by $\mathrm{d}\xi$, to the directional derivative

$$\frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{u}} = \sum_{i=1}^{N}\left(\frac{\partial f}{\partial x_i}\right)_{x_{j\neq i}}u_i = \boldsymbol{u}\cdot\nabla f . \tag{67}$$

Repeating the previous steps—taking the total differential and replacing the $\mathrm{d}x_i$ by $\mathrm{d}\xi$—yields

$$\frac{\mathrm{d}^2 f}{\mathrm{d}\boldsymbol{u}^2} = \sum_{i=1}^{N}\sum_{j=1}^{N}\left(\frac{\partial^2 f}{\partial x_i\,\partial x_j}\right)u_i u_j \tag{68}$$

For $N=2$ (the generalization to a larger number of arguments is straightforward) we can now set

$$\begin{aligned}
\boldsymbol{u} &= \begin{pmatrix}1\\0\end{pmatrix} &&\Rightarrow f_{10} = \frac{\partial^2 f}{\partial \boldsymbol{u}^2} = \left(\frac{\partial^2 f}{\partial x_1^2}\right)\\
\boldsymbol{u} &= \begin{pmatrix}0\\1\end{pmatrix} &&\Rightarrow f_{01} = \frac{\partial^2 f}{\partial \boldsymbol{u}^2} = \left(\frac{\partial^2 f}{\partial x_2^2}\right)\\
\boldsymbol{u} &= \begin{pmatrix}1\\1\end{pmatrix} &&\Rightarrow f_{11} = \frac{\partial^2 f}{\partial \boldsymbol{u}^2} = \left(\frac{\partial^2 f}{\partial x_1^2}\right)+\left(\frac{\partial^2 f}{\partial x_1\,\partial x_2}\right)\left(\frac{\partial^2 f}{\partial x_2^2}\right)
\end{aligned} \tag{69}$$

and obtain the mixed derivative as

$$\left(\frac{\partial^2 f}{\partial x_1\,\partial x_2}\right) = \frac{1}{2}\left(f_{11}-f_{10}-f_{01}\right) \tag{70}$$

The three second-order derivatives can be computed with Eq. (61), using ξ as (single) argument with three different $\boldsymbol{u}$ vectors. It should be noted that Eq. (70) is not a finite-difference differentiation formula and therefore does not necessarily suffer from cancellation errors.

9. Appendix D: Python example

This example in the python programming language (version 3.8) with the van der Waals EOS uses the `multicomplex` Python package, version 0.12

```
import numpy as np
from scipy.special import factorial

# Can be installed with: pip install multicomplex
# Or download from https://github.com/usnistgov/multicomplex
import multicomplex as mcx

R = 8.314462618 # J/mol/K

# Values for argon
Tc = 150.687 # K
pc = 4863000.0 # Pa

class vdWEOS:
    a = (27/64)*(R*Tc)**2/pc
    b = (1/8)*(R*Tc)/pc

    def alphar(self, T, rho):
        return -np.log(1.0-self.b*rho)-self.a*rho/(R*T)

# Instance of the class
vdW = vdWEOS()

# This state point
T = 300 # K
rho = 1.3 # mol/m^3

# Pressure
dalphardrho = mcx.diff_mcx1(lambda r: vdW.alphar(T, r), rho, 1, True)[1]
p = rho*R*T*(1+rho*dalphardrho)
print(f'p: {p} Pa')

# The first few virial coefficients
# Derivatives of alphar at zero density up to fourth order
ders = mcx.diff_mcx1(lambda r: vdW.alphar(T, r), 0.0, 4, True)
for n in range(2,5):
    B_n = ders[n-1]/factorial(n-2)
    units = f'm^{3*(n-1)}/mol^{(n-1)}'
    print(f'B_{n}: {B_n} {units}')

# Isochoric tension
# First derivative of alphar w.r.t. T at constant density
```

```
dalphardT = mcx.diff_mcx1(lambda T_: vdW.alphar(T_, rho), T, 1, True)[1]
# Cross (density, temperature) derivative of alphar
d2alphardTdrho = mcx.diff_mcxN(lambda x: vdW.alphar(x[0], x[1]), [T, rho], [1, 1])
betaV = (1 + rho*dalphardT + rho*T*d2alphardTdrho)*(rho*R)
print(f'betaV: {betaV} Pa/K')

"""
yields the output:
p: 3242.546045484618 Pa
B_2: -2.238945068494697e-05 m^3/mol^1
B_3: 1.0371257814774649e-09 m^6/mol^2
B_4: 3.340005219645591e-14 m^9/mol^3
betaV: 10.809571850439895 Pa/K
"""
```

Acknowledgments

We are grateful to Brad Alpert (National Institute of Standards and Technology, Boulder, CO, USA) for valuable advice.

10. References

[1] Dieterici C (1899) Ueber den kritischen Zustand. *Annalen der Physik* 69:685–705. https://doi.org/10.1002/andp.18993051111
[2] Sadus RJ (2001) Equations of state for fluids: The Dieterici approach revisited. *Journal of Chemical Physics* 115:1460–1462. https://doi.org/10.1063/1.1380711
[3] Sadus RJ (2002) Erratum: "Equations of state for fluids: The Dieterici approach revisited." *Journal of Chemical Physics* 116:5913. https://doi.org/10.1063/1.1455624
[4] Boshkova OL, Deiters UK (2010) Soft repulsion and the behavior of equations of state at high pressures. *International Journal of Thermophysics* 31:227–252. https://doi.org/10.1007/s10765-010-0727-7
[5] Jackson G, Chapman WG, Gubbins KE (1988) Phase equilibria of associating fluids. Spherical molecules with multiple bonding sites. *Molecular Physics* 65:1–31. https://doi.org/10.1080/00268978800100821
[6] Chapman WG, Jackson G, Gubbins KE (1988) Phase equilibria of associating fluids. Chain molecules with multiple bonding sites. *Molecular Physics* 65:1057–1079. https://doi.org/10.1080/00268978800101601
[7] Leal AMM (2018) autodiff, a modern, fast and expressive C++ library for automatic differentiation. Available at https://autodiff.github.io
[8] Jordan-Engeln G, Reutter F (1985) *Numerische Mathematik für Ingenieure* (Bibliographisches Institut, Mannheim).
[9] Press WH, Teukolsky SA, Vetterling WT, Flannery BP (2002) *Numerical Recipes in C* (Cambridge University Press, Cambridge).
[10] Lyness JN, Moler CB (1967) Numerical differentiation of analytic functions. *SIAM Journal on Numerical Analysis* 4:202–210. https://doi.org/10.1137/0704019
[11] Deiters UK, Bell IH (2020) Differential equations for critical curves of fluid mixtures. *Industrial and Engineering Chemistry Research* 59:19061–19076. https://doi.org/10.1021/acs.iecr.0c03667
[12] Squire W, Trapp G (1998) Using complex variables to estimate derivatives of real functions. *SIAM Review* 40:110–112. https://doi.org/10.1137/S003614459631241X
[13] Higham N (2018) Differentiation with(out) a difference. *SIAM News* 51(5):2.
[14] Lantoine G, Russell RP, Dargent T (1998) Using Multicomplex Variables for Automatic Computation of High-order Derivatives. *ACM Transactions on Mathematical Software* 38:16. http://doi.acm.org/10.1145/2168773.2168774
[15] Casado JMV, Hewson R (2020) Algorithm 1008: Multicomplex Number Class for Matlab, with a Focus on the Accurate Calculation of Small Imaginary Terms for Multicomplex Step Sensitivity Calculations. *ACM Transactions on Mathematical Software* 46:2. https://doi.org/10.1145/3378542
[16] Bornemann F (2010) Accuracy and Stability of Computing High-order Derivatives of Analytic Functions by Cauchy Integrals. *Foundations of Computational Mathematics* 11:1–63. https://doi.org/10.1007/s10208-010-9075-z
[17] IEEE (2019) *754-2019 – IEEE Standard for Floating-Point Arithmetic*. https://doi.org/10.1109/IEEESTD.2019.8766229

[18] van Konynenburg PH, Scott RL (1980) Critical lines and phase equilibria in binary van der Waals mixtures. *Philosophical Transactions of the Royal Society A* 298:495–540. https://doi.org/10.1098/rsta.1980.0266

[19] Kraska T, Deiters UK (1992) Systematic investigation of the phase behavior in binary fluid mixtures. II. Calculations based on the Carnahan–Starling–Redlich–Kwong equation of state. *Journal of Chemical Physics* 96:539–547. https://doi.org/10.1063/1.462490

[20] Boshkov LZ (1992) Bifurcations—a possibility to generalize the thermodynamic description of phase diagrams of two-component fluids. *Ber Bunsenges Phys Chem* 96:940–943.

[21] Bell IH (2019) multicomplex (multicomplex algebra library in c++ and python). https://doi.org/10.18434/mds2-2482

[22] Verheyleweghen A (2014) Computation of higher-order derivatives using the multi-complex step method (NTNU). Available at http://folk.ntnu.no/preisig/HAP_Specials/AdvancedSimulation_files/2014/AdvSim-2014__Verheule_Adrian_Complex_differenetiation.pdf.

[23] Directional/derivative. Available at https://en.wikipedia.org/wiki/Directional_derivative. Accessed on 2018-02-21.

About the authors: *Ulrich Deiters studied chemistry at the Ruhr University in Bochum, Germany, where he obtained the doctorate of natural sciences in 1979. After a postdoctoral year at the Cornell University he returned to Bochum to found his own research group. From 1993 to his formal retirement in 2018 he was a professor of physical chemistry at the University of Cologne, Germany. He was chairman of the Subcommittee of Thermodynamic Data of the IUPAC Commission I.2 on Thermodynamics from 1997 to 2002, then until 2008 a member of the board of directors of the IACT (International Association for Chemical Thermodynamics).*

Ian Bell obtained his Bachelor of Sciences in Mechanical Engineering from Cornell University in 2006 and his doctor of philosophy from Purdue University in 2011. He was a National Research Council postdoctoral researcher in the Applied Chemicals and Materials Division of the Materials and Measurements Laboratory of NIST, and has been a staff mechanical engineer in that division since 2017. He conducts research in the theory and modeling of the thermophysical properties of pure fluids and mixtures.

The National Institute of Standards and Technology is an agency of the U.S. Department of Commerce.