Bergische Universität
Wuppertal

Real and Complex Taylor Arithmetic in C-XSC

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Preprint 2005/4

Wissenschaftliches Rechnen/
Softwaretechnologie

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich C (Mathematik und Naturwissenschaften) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadressen

Frithjof Blomquist
Adlerweg 6
D-66436 Püttlingen

E-mail: `blomquist@math.uni-wuppertal.de`

Werner Hofschuster
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `hofschuster@math.uni-wuppertal.de`

Walter Krämer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `kraemer@math.uni-wuppertal.de`

Real and Complex Taylor Arithmetic in C-XSC

Frithjof Blomquist, Werner Hofschuster, Walter Krämer

Contents

Abstract

In **C-XSC** the Taylor Arithmetic is implemented for real as well as for complex variables. For a given function $f : \mathbb{R} \to \mathbb{R}$ or $f : \mathbb{R}^2 \to \mathbb{R}$ composed of the elementary **C-XSC** functions together with the arithmetic operators, the Taylor coefficients or the (partial) derivatives up to a given order p can be computed using the IEEE format or in higher precision using the interval staggered format. Additionally the Taylor Arithmetic is implemented in the IEEE Format for one complex variable. Each of the described Taylor Arithmetics has a **C++** class of its own, and their elements and member functions are detailed together with simple sample programs. With these tools the user has an easy access for solving appropriate numerical problems.

Keywords: Taylor arithmetic, guaranteed computations, staggered arithmetic, complex Taylor arithmetic, automatic differentiation, complex interval arithmetic, mathematical software, C-XSC

1 Introduction

The computation of derivatives is a task which one encounters in numerical analysis very often. Whenever a Newton method is to be applied then the derivative or the Jacobian matrix of the functions involved in the problem must be computed somehow. Also optimization methods use gradients or even Hesse matrices.

Many numerical textbooks consider it as difficult to compute such derivatives. Numerical libraries on computers usually put the burden of supplying subroutines for derivatives to the user of these libraries. Alternatively, derivatives are often replaced by finite difference quotients, which leads to many unnecessary additional problems like truncation and cancellation errors. Because of these apparent difficulties many numerical textbooks do not even try to consider methods which need derivatives of even higher order, like methods based on Taylor expansions.

Sometimes the use of symbolic differentiation is recommended to produce expressions for the needed derivatives. Almost all computer algebra systems can be utilized for this purpose. Many of them have features which allow the derived expressions to be converted to the syntax of some programming language. However, the expressions of the derivatives obtained in this way are often extremely complex and therefore costly to evaluate and sometimes they may even be numerically unstable to evaluate.

For many years already there exist interesting and very efficient methods to compute derivative values *exactly* (if exact arithmetic is used) for a large class of functions, i.e. in principle all functions which can be programmed in a common programming language. These methods are known as *automatic differentiation* and were developed already since the 60's by e.g. Moore [33], and later by Rall [35] - [41], Griewank [17], [18], Iri [25], Corliss [11], [12] Fischer, [13] - [15], Berz [6], Jerell [26] and many others. Automatic differentiation is now recognized as a very useful tool by a wider audience in the community of numerical analysts. An extensive bibliography, compiled by G. Corliss is contained in [18].

The main difference between these methods and finite differences and symbolic differentiation is that automatic differentiation computes derivative *values* which are the *exact* values of the derivatives (if exact arithmetic is used in the computation). I.e. automatic differentiation is a numerical method just as finite differences but in contrast to the latter it produces *exact* results. On the other side symbolic differentiation also produces exact results, however, their *expressions* for the derivatives are calculated. Numerical *values* of the derivatives are then obtained by evaluating the expressions numerically for suitable parameters. This indirect way often makes symbolic differentiation much more costly. Automatic differentiation can be viewed as a combination of symbolic differentiation and numerical evaluation performed simultaneously. I.e. the differentiation rules are applied to the given function but they are applied immediately to the numerical values not first to the expressions. This is the reason why automatic differentiation is exact just as symbolic differentiation but at the same time more efficient.

Also, during the last years many variants of automatic differentiation have been developed which differ primarily in the order in which the differentiation rules are applied. We can mainly distinguish between two different types of methods which are called the *foreward mode* and the *reverse mode* of automatic differentiation.

Historically the forward mode was used first whereas the reverse mode was developed later. For functions in several variables the reverse mode is usually much faster than the forward mode, the difference being very drastic: for a function $f : \mathbb{R}^n \to \mathbb{R}$ the time for computing the gradient using the forward mode is proportional to n times the time required for an evaluation of f. In contrast, for the reverse mode the time to compute the gradient is less than about 5 times the cost for an evaluation of f, see e.g. [17], [25], [13], [16]. Amazingly, this time ratio is *independent* of the dimension n !

In section 2 we will first discuss the one-dimensional case with higher derivatives, i.e. we present an automatic differentiation method for the computation of Taylor coefficients. This method is a forward method.

2 Functions in One Real Variable

2.1 Introduction

The computation of high derivatives of a function is often considered to be a tedious and error-prone task which is the reason why there exist almost no numerical methods making use of higher derivatives. Only the Taylor series method for the approximate solution of ordinary differential equations is sometimes mentioned in textbooks. However, this method is usually discarded immediately because it is considered to be far too expensive. Higher derivatives often occur also in remainder terms of various numerical approximation methods such as Taylor expansion, interpolation, numerical integration and the like.

In this section we show that the computation of such higher derivatives is indeed very easy and efficient and that even unknown intermediate values which often appear in remainder terms can be handled almost trivially by the use of interval arithmetic. Here the combination of automatic differentiation and interval arithmetic has a strong effect of synergy: Computing high derivatives efficiently and enclosing unknown intermediate

points in intervals results in methods of totally new quality. This combination makes it possible to estimate remainder terms rigorously and thus to derive proofs automatically on a computer which were not possible earlier or which were at least much more complicated.

2.2 Theoretical Background

To keep the presentation simple assume that $f : \mathbb{R} \to \mathbb{R}$ is an analytic function. Then f can be expanded in a Taylor series around a point x_0

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k. \tag{1}$$

We introduce the following short notation for the Taylor coefficient $\dfrac{f^{(k)}(x_0)}{k!}$ of f at the point of expansion $x = x_0$

$$(f)_k := \frac{f^{(k)}(x_0)}{k!} = \frac{1}{k!}\frac{d^k f(x_0)}{dx^k}. \tag{2}$$

Using (2) we can rewrite (1) as

$$f(x) = \sum_{k=0}^{\infty} (f)_k (x - x_0)^k. \tag{3}$$

Now let us consider functions f which are the composition of arithmetic operations and elementary functions. For functions of such structures we can recursively determine the Taylor coefficients by applying appropriate rules for each single operation occurring in the expression of f.
Starting with the two trivial cases we state that for a constant function $f = c$

$$(c)_0 = c, \quad (c)_k = 0, \text{ if } k \geq 1 \tag{4}$$

and for the independent variable $f = x$:

$$(x)_0 = x, \quad (x)_1 = 1, \quad (x)_k = 0, \text{ if } k \geq 2. \tag{5}$$

Due to the uniqueness of power series we immediately see that for the two functions $u(x)$ and $v(x)$ we have

$$(u \pm v)_k = (u)_k \pm (v)_k. \tag{6}$$

$$(u \cdot v)_k = \sum_{j=0}^{k} (u)_j (v)_{k-j} = \sum_{j=0}^{k} (u)_{k-j} (v)_j. \tag{7}$$

$$(u/v)_k = \frac{1}{(v)_0}\left[(u)_k - \sum_{j=1}^{k} (v)_j (u/v)_{k-j}\right]. \tag{8}$$

Here the rule of the quotient (8) can easily be obtained as follows: writing $f = u/v$, then $u = v \cdot f$, and the product rule (7) delivers

$$(u)_k = \sum_{j=0}^{k} (v)_j (f)_{k-j} = (v)_0 (f)_k + \sum_{j=1}^{k} (v)_j (f)_{k-j},$$

which must be solved for $(f)_k = (u/v)_k$ to obtain the quotient rule (8).

Now let us consider functions $f(u)$ and let $u = u(x)$ be an analytic function of x. With these notations we have

$$w(x) \equiv f(u(x)) \tag{9}$$

$$\frac{d}{dx} w = w'(x) = \frac{d\, f(u)}{du}\bigg|_{u=u(x)} \cdot u'(x) = f'(u)\bigg|_{u=u(x)} \cdot u'(x); \tag{10}$$

As a first application let us consider the square root $f(u) = \sqrt{u}$ which can be written in the form $f \cdot f = u$. Applying the product rule (7) to $f \cdot f$ after some simple conversions, for $k \geq 1$, we find

$$\begin{aligned} (u)_k &= (f \cdot f)_k = \sum_{j=0}^{k} (f)_j (f)_{k-j} \\ &= (f)_0 (f)_k + \sum_{j=1}^{k-1} (f)_j (f)_{k-j} + (f)_k (f)_0 \\ &= 2(f)_0 (f)_k + \sum_{j=1}^{k-1} (f)_j (f)_{k-j}. \end{aligned}$$

Solving the last equation for $(f)_k$, we get:

$$(f)_k = \frac{1}{2(f)_0} \left[(u)_k - \sum_{j=1}^{k-1} (f)_j (f)_{k-j} \right].$$

Be aware that $(f)_j$ is the Taylor coefficient with respect to x, i.e.

$$(f)_j := \frac{1}{j!} \frac{d^j}{d\, x^j} f\left(u(x)\right) \bigg|_{u=u(x_0)}$$

Substituting f with $\sqrt{u}$ for the square root we get

$$(\sqrt{u})_k = \frac{1}{2(\sqrt{u})_0} \left[(u)_k - \sum_{j=1}^{k-1} (\sqrt{u})_j (\sqrt{u})_{k-j} \right], \quad k \geq 1. \tag{11}$$

The recursive construction in (11) is obvious, i.e. for calculating $(\sqrt{u})_k$ all Taylor coefficients $(\sqrt{u})_j$ of order $j < k$ must be calculated before. The sum in (11) consists of equal summands by twos, so we get the more effective relation

$$(\sqrt{u})_k = \begin{cases} \dfrac{1}{2(\sqrt{u})_0}\left[(u)_k - 2\sum_{j=1}^{(k-1)/2}(\sqrt{u})_j(\sqrt{u})_{k-j}\right], & \text{if } k \text{ odd} \\ \dfrac{1}{2(\sqrt{u})_0}\left[(u)_k - 2\sum_{j=1}^{(k-2)/2}(\sqrt{u})_j(\sqrt{u})_{k-j} - (\sqrt{u})_{k/2}^2\right], & \text{if } k \text{ even,} \end{cases}$$

where the last form uses only about half as many operations as the form in (11). Due to $(f(u)\,)_0 = f((u)_0)$ of course we have $(\sqrt{u})_0 = \sqrt{(u)_0}$.

Also for the square function $f(u) = u^2$ we can write down a more economical form than just for the multiplication in (7):

$$(u^2)_k = \begin{cases} 2\sum_{j=0}^{(k-1)/2}(u)_j(u)_{k-j}, & \text{if } k \text{ odd} \\ 2\sum_{j=0}^{(k-2)/2}(u)_j(u)_{k-j} + (u)_{k/2}^2 & \text{if } k \text{ even.} \end{cases}$$

To find similar recursion formulas also for elementary functions, we need a further relationship between the Taylor coefficients of f and its derivative f'. Differentiating (3) with respect to x we get a series for f':

$$f'(x) = \sum_{k=1}^{\infty} k \cdot (f)_k (x - x_0)^{k-1} = \sum_{k=0}^{\infty} (k+1)(f)_{k+1}(x - x_0)^k.$$

On the other hand, in (3) we can replace f by f':

$$f'(x) = \sum_{k=0}^{\infty} (f')_k (x - x_0)^k.$$

Equating coefficients of these two representations yields

$$\boxed{(f')_k = (k+1) \cdot (f)_{k+1}} \tag{12}$$

and (12) keeps valid, even if f is given by $f = f(u(x))$. In this case f' is defined by

$$f' := \left.\frac{d\,f(u)}{du}\right|_{u=u(x)} \cdot u'(x)$$

The relationship (12) is very important in deriving new recursion formulas for elementary functions and also for other applications like the Taylor series solution of ordinary differential equations.

As a first application of (12) we consider the exponential function $f(u) = e^u$. Calculating the first derivative of $w(x) = f(u(x)) = e^{u(x)}$ with respect to x it follows $w' = w \cdot u'$, and the rule of the product delivers

$$(w')_{k-1} = (w \cdot u')_{k-1} = \sum_{j=0}^{k-1} (w)_j (u')_{k-1-j}, \quad k \geq 1.$$

With (12) it holds $(w')_{k-1} = k \cdot (w)_k$ resp. $(u')_{k-1-j} = (k-j)(u)_{k-j}$, and we get

$$k \cdot (w)_k = \sum_{j=0}^{k-1} (w)_j (k-j)(u)_{k-j}, \quad k \geq 1.$$

Solving this for $(w)_k = (e^u)_k$ we get the recursion formula of the exponential function:

$$\begin{aligned} (e^u)_k &= \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(e^u)_j (u)_{k-j}, \quad u = u(x), \quad k \geq 1 \\ (e^u)_0 &= e^{(u)_0}, \quad k = 0. \end{aligned}$$

In a very similar way we can obtain formulas for the Taylor coefficients of $\sin(u)$ and $\cos(u)$:

$$\begin{aligned} (\sin u)_k &= \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\cos u)_j (u)_{k-j}, \\ (\cos u)_k &= -\frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\sin u)_j (u)_{k-j}, \end{aligned}$$

and also for $\sinh(u)$ and $\cosh(u)$:

$$\begin{aligned} (\sinh u)_k &= \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\cosh u)_j (u)_{k-j}, \\ (\cosh u)_k &= \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\sinh u)_j (u)_{k-j}. \end{aligned}$$

Obviously these formulas have to be applied pair-wise.

For the power function $f(u) = u^a$ with a constant exponent a we can use the same technique. Calculating the first derivative of $w(x) = f(u(x)) = u(x)^a$ with respect to x it follows $w' \cdot u = a \cdot w \cdot u'$, and the rule of the product (7) delivers

$$(u^a)_k = \frac{1}{k \cdot (u)_0} \sum_{j=0}^{k-1} [a \cdot (k-j) - j] \cdot (u^a)_j \cdot (u)_{k-j}, \quad k \geq 1.$$

Almost all the other elementary functions which are implemented in C-XSC can be treated in the following way. For $w(x) = f(u(x))$ the first derivative with respect to x is given by

$$\frac{d}{dx}w = w'(x) = \left.\frac{d\,f(u)}{du}\right|_{u=u(x)} \cdot u'(x) = f'(u)|_{u=u(x)} \cdot u'(x).$$

Writing $f'(u)|_{u=u(x)} = \left.\frac{1}{g(u)}\right|_{u=u(x)}$ or more simple $f'(u) = \frac{1}{g(u)}$, it follows

(13)
$$\boxed{u' = g \cdot w'}$$

Together with (12) the rule of the product (7) delivers

$$\begin{aligned} k(u)_k &= (u')_{k-1} = (g \cdot w')_{k-1} = \sum_{j=0}^{k-1}(g)_{k-j-1}(w')_j \\ &= \sum_{j=0}^{k-1}(g)_{k-j-1}(j+1)(w)_{j+1} = \sum_{j=1}^{k} j \cdot (g)_{k-j} \cdot (w)_j \\ &= \sum_{j=1}^{k-1} j \cdot (w)_j \cdot (g)_{k-j} + k \cdot (w)_k \cdot (g)_0 \quad \rightsquigarrow \end{aligned}$$

$$(w)_k = \frac{1}{(g)_0}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j \cdot (w)_j \cdot (g)_{k-j}\right], \quad k \geq 1.$$

As the Taylor coefficients of order k refer to the independent variable x, it holds $(w)_k = (f(u))_k = (f)_k$, and finally we get:

(14)
$$\boxed{\begin{aligned} (f)_k &= \frac{1}{(g)_0}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j \cdot (f)_j \cdot (g)_{k-j}\right], \quad k \geq 1 \\ f(u) &= f(u(x)); \quad g(u) = \left[\frac{d\,f(u)}{du}\right]^{-1}, \quad \frac{d\,f(u)}{du} \neq 0 \end{aligned}}$$

Using the relation (14) the Taylor coefficients of the remaining functions can be listed in the following table. $g(u)$ is defined by (14).

Recursion Formulas of the Taylor Coefficients $(f)_k$, (part 1)		
$f(u)$	$g(u)$	$(f)_k,\ k \geq 1$
$\ln u$	u	$\frac{1}{(u)_0}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j \cdot (\ln u)_j (u)_{k-j}\right]$
$\tan u$	$\cos^2 u$	$\frac{1}{\cos^2(u)_0}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j \cdot (\tan u)_j (\cos^2 u)_{k-j}\right]$
$\cot u$	$-\sin^2 u$	$\frac{-1}{\sin^2(u)_0}\left[(u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j \cdot (\cot u)_j (\sin^2 u)_{k-j}\right]$
$\arcsin u$	$\sqrt{1-u^2}$	$\frac{1}{\sqrt{1-(u)_0^2}}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j (\arcsin u)_j (\sqrt{1-u^2})_{k-j}\right]$
$\arccos u$	$-\sqrt{1-u^2}$	$\frac{-1}{\sqrt{1-(u)_0^2}}\left[(u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j (\arccos u)_j (\sqrt{1-u^2})_{k-j}\right]$
$\arctan u$	$1+u^2$	$\frac{1}{1+(u)_0^2}\left[(u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j (\arctan u)_j (1+u^2)_{k-j}\right]$
$\operatorname{arccot} u$	$-(1+u^2)$	$\frac{-1}{1+(u)_0^2}\left[(u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j (\operatorname{arccot} u)_j (1+u^2)_{k-j}\right]$

Recursion Formulas of the Taylor Coefficients $(f)_k$, (part 2)		
$f(u)$	$g(u)$	$(f)_k,\ k \geq 1$
$\tanh u$	$\cosh^2 u$	$\dfrac{1}{\cosh^2(u)_0}\left[(u)_k - \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\tanh u)_j(\cosh^2 u)_{k-j}\right]$
$\coth u$	$-\sinh^2 u$	$\dfrac{-1}{\sinh^2(u)_0}\left[(u)_k + \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\coth u)_j(\sinh^2 u)_{k-j}\right]$
$\operatorname{arsinh} u$	$\sqrt{u^2+1}$	$\dfrac{1}{\sqrt{(u)_0^2+1}}\left[(u)_k - \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\operatorname{arsinh} u)_j(\sqrt{u^2+1})_{k-j}\right]$
$\operatorname{arcosh} u$	$\sqrt{u^2-1}$	$\dfrac{1}{\sqrt{(u)_0^2-1}}\left[(u)_k - \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\operatorname{arsinh} u)_j(\sqrt{u^2-1})_{k-j}\right]$
$\operatorname{artanh} u$	$1-u^2$	$\dfrac{1}{1-(u)_0^2}\left[(u)_k - \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\operatorname{artanh} u)_j(1-u^2)_{k-j}\right]$
$\operatorname{arcoth} u$	$1-u^2$	$\dfrac{1}{1-(u)_0^2}\left[(u)_k - \dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1} j(\operatorname{arcoth} u)_j(1-u^2)_{k-j}\right]$
u^2	$(u^2)_k = \displaystyle\sum_{j=0}^{k}(u)_j(u)_{k-j}$	
$\sqrt{u}$	$(\sqrt{u})_k = \dfrac{1}{2(\sqrt{u})_0}\left[(u)_k - \displaystyle\sum_{j=1}^{k-1}(\sqrt{u})_j(\sqrt{u})_{k-j}\right]$	
$\sin u$	$(\sin u)_k = \dfrac{1}{k}\displaystyle\sum_{j=0}^{k-1}(k-j)(\cos u)_j(u)_{k-j}$	
$\cos u$	$(\cos u)_k = -\dfrac{1}{k}\displaystyle\sum_{j=0}^{k-1}(k-j)(\sin u)_j(u)_{k-j}$	

Recursion Formulas of the Taylor Coefficients $(f)_k$, (part 3)	
$f(u)$	$(f)_k,\ k \geq 1$
$\sinh u$	$(\sinh u)_k = \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\cosh u)_j(u)_{k-j}$
$\cosh u$	$(\cosh u)_k = \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\sinh u)_j(u)_{k-j}$
e^u	$(e^u)_k = \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(e^u)_j(u)_{k-j}$
2^u	$(2^u)_k = \frac{\ln 2}{k}\sum_{j=0}^{k-1}(k-j)(2^u)_j(u)_{k-j}$
10^u	$(10^u)_k = \frac{\ln 10}{k}\sum_{j=0}^{k-1}(k-j)(10^u)_j(u)_{k-j}$
$\log_2 u$	$(\log_2 u)_k = \frac{1}{\ln 2}\cdot(\ln u)_k$
$\log_{10} u$	$(\log_{10} u)_k = \frac{1}{\ln 10}\cdot(\ln u)_k$

Remarks:

- For $(u^2)_k$ and $(\sqrt{u})_k$ there are more efficient recursion formulas, already listed on page 8.
- The recursion formulas for sin u and cos u must be applied pair-wise. Analogously the formulas for sinh u and cosh u are to be treated.

To get inclusions of the Taylor coefficients, the recursion formulas are to by evaluated with interval arithmetic, and so some interval overestimations are unavoidable. However, if the point of expansion lies near a zero of some derivative, these overestimations increase dramatically. In these cases the recursion formulas must be transformed in such a way that the desribed overestimations are minimized to a tolerable measure.
As a **first example** let us consider the function

$$\begin{aligned} w(x) &= f(u(x)) = u \cdot \ln u \quad \text{with} \quad u = u(x); \\ w'(x) &= (1 + \ln u) \cdot u'(x) \end{aligned}$$

For $u(x_0) = x_0 = 1/e$ the expression $(1+\ln u)$ has a zero, so in their environment $w'(x)$ will be evaluated with severe cancellation effects. However, if a function like $\ln(e \cdot u) \equiv 1 + \ln u$ is implemented in such a way that the described cancellations are sufficiently reduced, then $w'(x)$ can tightly be included, even near $x_0 = 1/e$, so we write

$$w'(x) = \ln(e \cdot u) \cdot u'(x).$$

With formula (12) and the rule of product (7) it holds

$$\begin{aligned} k \cdot (w)_k &= (w')_{k-1} = (\ln(e \cdot u) \cdot u')_{k-1} \\ &= \sum_{j=0}^{k-1} (\ln(e \cdot u))_j (k-j)(u)_{k-j}, \end{aligned}$$

and with $(w)_k = (u \cdot \ln u)_k$ we finally get the recursion formula

$$(u \cdot \ln u)_k = \frac{1}{k} \sum_{j=0}^{k-1} (k-j) \cdot (\ln(e \cdot u))_j \cdot (u)_{k-j}, \quad k \geq 1.$$

As a **next example** we consider the power function

$$\begin{aligned} w(x) &= f(u(x)) = u^u = e^{u \cdot \ln(u)} \quad \text{with} \quad u = u(x) > 0; \\ w'(x) &= \ln(e \cdot u) \cdot e^{u \cdot \ln(u)} \cdot u'(x) \end{aligned}$$

Again, with formula (12) and the rule of product (7), it holds

$$\begin{aligned} k \cdot (w)_k &= (w')_{k-1} = (\ \ln(e \cdot u) \cdot e^{u \cdot \ln(u)} \cdot u'\ )_{k-1} \\ &= \sum_{j=0}^{k-1} (\ \ln(e \cdot u) \cdot e^{u \cdot \ln(u)}\ )_j (k-j)(u)_{k-j} \end{aligned}$$

and with $(w)_k = (u^u)_k$ we get the recursion formula

$$(u^u)_k = \frac{1}{k} \sum_{j=0}^{k-1} (k-j) \cdot (\ \ln(eu) \cdot e^{u \cdot \ln(u)}\ )_j \cdot (u)_{k-j}, \quad u(x) > 0, \quad k \geq 1.$$

As a **next example** we consider the function

$$w(x) = f(u(x)) = \sqrt{1+u(x)} - 1, \quad w+1 = \sqrt{1+u},$$
$$w \cdot w = u - 2w,$$

Then, applying the rule of product, we get

$$(w \cdot w)_k = \sum_{j=0}^{k} (w)_j \cdot (w)_{k-j} = (u)_k - 2(w)_k$$
$$(w)_0 \cdot (w)_k + \sum_{j=1}^{k-1} (w)_j \cdot (w)_{k-j} + (w)_k \cdot (w)_0 = (u)_k - 2(w)_k$$
$$2(w)_0 \cdot (w)_k + 2(w)_k = (u)_k - \sum_{j=1}^{k-1} (w)_j \cdot (w)_{k-j} \quad \rightsquigarrow$$
$$(f)_k = \frac{1}{2\sqrt{1+(u)_0}} \left[(u)_k - \sum_{j=1}^{k-1} (f)_j (f)_{k-j} \right], \quad (f)_j = (\sqrt{1+u} - 1)_j.$$

As a **next example** we consider the function

$$w(x) = f(u(x)) = \sqrt{1-u^2(x)}, \qquad u^2(x) < 1$$
$$w \cdot w = 1 - u^2.$$

The rule of product (7) delivers

$$(w \cdot w)_k = (1-u^2)_k = -(u^2)_k, \quad k \geq 1$$
$$\sum_{j=0}^{k} (w)_j \cdot (w)_{k-j} = -(u^2)_k$$
$$2(w)_0 \cdot (w)_k + \sum_{j=1}^{k-1} (w)_j \cdot (w)_{k-j} = -(u^2)_k \qquad \rightsquigarrow$$
$$(\sqrt{1-u^2})_k = \frac{-1}{2\sqrt{1-(u^2)_0}} \left[(u^2)_k + \sum_{j=1}^{k-1} (\sqrt{1-u^2})_j (\sqrt{1-u^2})_{k-j} \right], \; k \geq 1$$

For the function $w(x) = \sqrt{u^2(x) - 1}$ we analogously find the recursion formula

$$(\sqrt{u^2-1})_k = \frac{1}{2\sqrt{(u^2)_0 - 1}} \left[(u^2)_k - \sum_{j=1}^{k-1} (\sqrt{u^2-1})_j (\sqrt{u^2-1})_{k-j} \right], \; k \geq 1.$$

The recursion formulas of the last examples are listed in the table on page 16. Be aware that, for a shorter run time, the sums $\sum_{j=1}^{k-1} (f)_j \cdot (f)_{k-j}$ in the recursion formulas of the

root functions can be written in the following form, see page 8.

$$\sum_{j=0(1)}^{k-1}(f)_j\cdot(f)_{k-j}=\begin{cases}2\displaystyle\sum_{j=0(1)}^{(k-1)/2}(f)_j(f)_{k-j}, & \text{if } k \text{ odd},\\ 2\displaystyle\sum_{j=0(1)}^{(k-2)/2}(f)_j(f)_{k-j}+(u)_{k/2}^2, & \text{if } k \text{ even}.\end{cases}$$

Recusion Formulas of the Taylor Coefficients $(f)_k$, (part 4)	
$f(u)$	$(f)_k,\ k\geq 1$
$\ln(e\cdot u)$	$(\ln(e\cdot u))_k=\dfrac{1}{(u)_0}\left[(u)_k-\dfrac{1}{k}\displaystyle\sum_{j=1}^{k-1}j\cdot(\ln(e\cdot u))_j\cdot(u)_{k-j}\right]$
$u\cdot\ln(u)$	$(u\cdot\ln(u))_k=\dfrac{1}{k}\displaystyle\sum_{j=0}^{k-1}(k-j)\cdot(\ln(e\cdot u))_j\cdot(u)_{k-j}$
e^u-1	$(e^u-1)_k=\dfrac{1}{k}\displaystyle\sum_{j=0}^{k-1}(k-j)\cdot(e^u)_j\cdot(u)_{k-j}$
u^u	$(u^u)_k=\dfrac{1}{k}\displaystyle\sum_{j=0}^{k-1}(k-j)\cdot(\ \ln(eu)\cdot e^{u\cdot\ln(u)}\ )_j\cdot(u)_{k-j}$
$\sqrt{1+u}-1$	$(f)_k=\dfrac{1}{2\sqrt{1+(u)_0}}\left[(u)_k-\displaystyle\sum_{j=1}^{k-1}(f)_j\cdot(f)_{k-j}\right]$
$\sqrt{1-u^2}$	$(f)_k=\dfrac{-1}{2(f)_0}\left[(u^2)_k+\displaystyle\sum_{j=1}^{k-1}(f)_j\cdot(f)_{k-j}\right]$
$\sqrt{u^2-1}$	$(f)_k=\dfrac{1}{2(f)_0}\left[(u^2)_k-\displaystyle\sum_{j=1}^{k-1}(f)_j\cdot(f)_{k-j}\right]$

2.3 Implementation, IEEE Format

In C-XSC the `itaylor` arithmetic is declared in the module `itaylor.hpp` and implemented in `itaylor.cpp`. The **C++** class `itaylor` provides the following constructors:

Constructors of the class `itaylor`:

1. `itaylor();` // default constructor;
2. `itaylor(const itaylor& );` // copy constructor;
3. `itaylor(int order, const real& r);` // $(r, 1, 0, ..., 0)$;
4. `itaylor(int order, const interval& z);` // $(z, 1, 0, ..., 0)$;

If the third constructor is invoked by `itaylor t(5,0.1);` then the not representable decimal value 0.1 will first be rounded to the nearest IEEE number r_0 and after this a point interval $[r_0]$ is created, which does *not* includes the desired value 1/10. To avoid the so originated conversion error, the fourth constructor must be invoked by

```
itaylor t( 5,z );
```

and the interval `z` must be realized as in the sample program on page 51. Only then `z` delivers a thick interval, being an optimal inclusion of 1/10. The interval vector $(z, [1], [0], [0], [0], [0])$ with $5 + 1 = 6$ interval components is an attribute of the object `t`.

Initialization functions of the class `itaylor`:

1. `itaylor const_itaylor(int order, const real& r);`
2. `itaylor const_itaylor(int order, const interval& z);`
3. `itaylor var_itaylor(int order, const real& r);`
4. `itaylor var_itaylor(int order, const interval& z);`

With the assignments

```
itaylor c;   c = const_itaylor(6,0.25);
```

we get the interval vector $([0.25], [0], [0], [0], [0], [0], [0])$ with $6+1 = 7$ interval components. `c` is a constant value in the Taylor arithmetic. With the assignments

```
itaylor x;   x = var_itaylor(6,0.25);
```

we get the interval vector $([0.25], [1], [0], [0], [0], [0], [0])$ with $6+1 = 7$ interval components. `x` is now an independent variable in the Taylor arithmetic.
If the real values `r` are not presentable in the IEEE system then be aware of the described conversion errors and use the initialization functions number 2 or 4 instead of 1 or 3.

Assignment operators of the class `itaylor`:

1. `itaylor operator = ( const itaylor& t );`
2. `itaylor operator = ( int );`
3. `itaylor operator = ( const real& r );`
4. `itaylor operator = ( const interval& z );`

With the assignments

```
int p=6;   real r1, r2(3.5);   itaylor t(p,r1);   t = r2;
```

we get an interval vector $([3.5], [0], [0], [0], [0], [0], [0])$ with $6 + 1 = 7$ interval components, which is an attribute of the class object `t`. If the real value `r2` is not presentable in the IEEE system, again be aware of the described conversion errors.

Be aware that the assignment operators No. 2-4 **always** deliver a **constant** `itaylor` object and not an independent variable!

Access to the Taylor coefficients and derivatives; output procedure

1. `int get_order( const itaylor& t );`
2. `ivector get_all_coef( const itaylor& t );`
3. `interval get_j_coef( const itaylor& t, int j );`
4. `interval get_j_derive( const itaylor& t, int j );`
5. `void print_itaylor( const itaylor& t );`

The easiest way to explain the above functions is a small sample program. The Taylor coefficients and derivatives of the given polynomial

$$\begin{aligned} P_4(x) &= 2x^4 + x^3 + 4x^2 - 3x + 2 \\ &= (((2x + 1) \cdot x + 4) \cdot x - 3) \cdot x + 2, \quad \text{(Horner's scheme)} \end{aligned}$$

are to be included up to the order $p = 5$ for given points of expansion $x_0 \in \mathbb{R}$.

```
// Sample program itayl_ex1.cpp;

#include "itaylor.hpp" // Header file of class itaylor
#include <iostream>    // Input | output

using namespace cxsc;
using namespace std;
using namespace taylor;

main()
{
 /*-------------------------------------------------------------
    Inclusions of the Taylor coefficients and derivatives up to
    order p=5 for      P(x) = 2x^4 + x^3 + 4x^2 - 3x +2      at
    points of expansion x0 included by the interval z.
  -------------------------------------------------------------*/

    int p = 5;    // Order of expansion
    interval z;   // Interval to include the point of expansion
    itaylor P;    // Default constructor No. 1.
    while(1) {
        cout << endl << "Inclusion of x0; [x0,x0] = ? "; cin >> z;
        itaylor x(p,z);  // Constructor No. 4,

        P = (((real(2.0)*x + 1) // Polynomial of order 4.
              *x + 4)
              *x - real(3.0))
              *x + real(2.0);

        cout << SetPrecision(16,16) << Scientific << endl;
        print_itaylor(P);  // Output of Taylor coefficients.

        ivector derivative(0,p); // interval vector with
        // components from index 0 up to the order p.

        for(int i=0;i<=p; i++)
            derivative[i] = get_j_derive(P,i); // Derivatives
        cout << "Inclusions of the derivatives up to order 5 "
             << endl;
        for(int i=0; i<=p; i++) // Output of the derivatives
            cout  << i<<"th derivative:  " << derivative[i] << endl;
    }
} // main
```

With the point of expansion `x0` $= 0.1$, included by the thick interval `z` $\ni$ `x0`, the sample program `itayl_test1.cpp` produces the following output:

```
  Inclusion of x0; [x0,x0] = ? [0.1,0.1]
```

```
Output itaylor of order 5
i  0  component: [+1.7411999999999998E+000,+1.7412000000000001E+000]
i  1  component: [-2.1620000000000009E+000,-2.1619999999999994E+000]
i  2  component: [+4.4199999999999981E+000,+4.4200000000000009E+000]
i  3  component: [+1.7999999999999998E+000,+1.8000000000000008E+000]
i  4  component: [+2.0000000000000000E+000,+2.0000000000000000E+000]
i  5  component: [-0.0000000000000000E+000,+0.0000000000000000E+000]

Inclusions of the derivatives up to order 5
0th derivative:  [+1.7411999999999998E+000,+1.7412000000000001E+000]
1th derivative:  [-2.1620000000000009E+000,-2.1619999999999994E+000]
2th derivative:  [+8.8399999999999963E+000,+8.8400000000000017E+000]
3th derivative:  [+1.0799999999999998E+001,+1.0800000000000005E+001]
4th derivative:  [+4.8000000000000000E+001,+4.8000000000000000E+001]
5th derivative:  [-0.0000000000000000E+000,+0.0000000000000000E+000]
```

Remarks:

- The program input `[0.1,0.1] RET` delivers a thick interval `z`, which is an optimal inclusion of the not presentable decimal number $1/10 \in$ `z`.
- The above inclusions of the Taylor coefficients and derivatives are not only guaranteed for $0.1 \in$ `z` but also for all other real numbers $x_0 \in$ `z`.

In the code line, beginning with `P = (((real(2.0)*x ...`, the polynomial $P_4(x)$ is evaluated by Horner's scheme using the overloaded operators for the arithmetic operations with objects of the class `itaylor`. For example, the algorithm of the operator $*$ for the multiplication of two `itaylor` operands `u=u(x),v=v(x)` is given by

Computation of the Taylor coefficients for $\mathtt{u} \cdot \mathtt{v}$

input: `itaylor u,v` of length $p+1$,
output: Enclosure of the Taylor coefficients of $\mathtt{u} \cdot \mathtt{v}$,

for $k := 0$ **to** p **do** $(w)_k := \diamond \sum_{j=0}^{k} (u)_j \cdot (v)_{k-j}$;
return w; ($\diamond$ denotes the interval hull of the sum.)

The returned `itaylor` object w is a $(p+1)$-tuple containing the inclusions of the Taylor coefficients of $\mathtt{u} \cdot \mathtt{v}$ up to the p-th order. The operators $+, -, /$ are implemented analogously. If only one operator is of type `itaylor`, then the type of the other operator can be any element of the set

{ `int, real, interval` }

and each of these three operands is interpreted as a **constant** value.

The elementary functions of type `itaylor`, implemented in C-XSC with the recursion formulas of page 10, see part 1-4, are listed in the following table.

Elementary Functions of the Class itaylor `itaylor u=u(x); interval a; int n;`			
Function	**C-XSC Name**	**Function**	**C-XSC Name**
u^2	`sqr(u)`	$\arcsin(u)$	`asin(u)`
$\sqrt{u}$	`sqrt(u)`	$\arccos(u)$	`acos(u)`
$\sqrt[n]{u}$	`sqrt(u,n)`	$\arctan(u)$	`atan(u)`
$\sqrt{1+u^2}$	`sqrt1px2(u)`	$\operatorname{arccot}(u)$	`acot(u)`
$\sqrt{1-u^2}$	`sqrt1mx2(u)`	$\sinh(u)$	`sinh(u)`
$\sqrt{u^2-1}$	`sqrtx2m1(u)`	$\cosh(u)$	`cosh(u)`
$\sqrt{1+u}-1$	`sqrtp1m1(u)`	$\tanh(u)$	`tanh(u)`
e^u	`exp(u)`	$\coth(u)$	`coth(u)`
e^u-1	`expm1(u)`	$\operatorname{arsinh}(u)$	`asinh(u)`
u^a	`pow(u,a)`	$\operatorname{arcosh}(u)$	`acosh(u)`
$\ln(u)$	`ln(u)`	$\operatorname{artanh}(u)$	`atanh(u)`
$\ln(1+u)$	`lnp1(u)`	$\operatorname{arcoth}(u)$	`acoth(u)`
$\sin(u)$	`sin(u)`		
$\cos(u)$	`cos(u)`		
$\tan(u)$	`tan(u)`		
$\cot(u)$	`cot(u)`		

The n-th root $\sqrt[n]{u}$ is defined for $(u)_0 \geq 0$ and $n \geq 1$; $\sqrt[1]{u} \equiv u$. Combining the above elementary functions with the arithmetic operators, we get expressions f of type `itaylor`, i.e. $(p+1)$-tuples with enclosures of the Taylor coefficients of f. With the following sample program we shew that an appropriate selection of the expression of f is essential for getting sufficient tight enclosures of the Taylor coefficients. For this purpose we consider the two equivalent functions $g(x) \equiv h(x)$

$$g(x) = \ln(1+e^x), \tag{15}$$

$$h(x) = x + \ln(1+e^{-x}). \tag{16}$$

The expressions $\ln(1+e^{\pm x})$ on the right-hand side are evaluated by the C-XSC function calls `lnp1(exp(±x))`.

```
/*------------------------------------------------------------
 Sample program itayl_ex2.cpp;
 Evaluation of the equivalent functions
        g(x) = ln(1+e^x)     h(x) = x + ln(1+e^(-x));
------------------------------------------------------------*/
#include "itaylor.hpp" // Header file of class itaylor
#include <iostream>    // Input | output

using namespace cxsc;
using namespace std;
using namespace taylor;
itaylor f1(const itaylor& u)
{
    itaylor w;
    interval z( get_j_coef(u,0) );
    if (Inf(z)>0) w = u + lnp1( exp(-u) );
    else w = lnp1( exp(u) );
    return w;
}
main() {
  int p = 4;    // Order of expansion
  itaylor f,g,h;  // Default constructor No. 1.
  itaylor x(p,interval(-706));

  g = lnp1( exp(x) );
  h = x + lnp1( exp(-x) );
  f = f1(x);
  cout << SetPrecision(16,16) << Scientific << endl;
  cout << "Taylor coefficients of g(x) are included by:" << endl;
  print_itaylor(g);  // Output of Taylor coefficients.
  cout << "Taylor coefficients of h(x) are included by:" << endl;
  print_itaylor(h);
  cout << "Taylor coefficients of f1(x) are included by:" << endl;
  print_itaylor(f);
}  // main
```

In code line `itaylor x(p,interval(-706));` we choose the point of expansion $x_0 = -706$ and the program produces the output:

```
Taylor coefficients of g(x) are included by:
Output itaylor of order 4
i  0  component: [2.443969469407074 9E-307,2.443969469407079 8E-307]
i  1  component: [2.443969469407074 5E-307,2.443969469407079 8E-307]
i  2  component: [1.221984734703537 0E-307,1.221984734703539 9E-307]
```

```
i  3  component: [4.0732824490117906E-308,4.0732824490117996E-308]
i  4  component: [1.0183206122529469E-308,1.0183206122529499E-308]

Taylor coefficients of h(x) are included by:
Output itaylor of order 4
i  0  component: [-4.5474735088646412E-013,7.9580786405131221E-013]
i  1  component: [-1.7763568394002505E-015,1.8873791418627662E-015]
i  2  component: [-1.7529495936443918E-015,1.8291647933680609E-015]
i  3  component: [-1.8101109934371439E-015,1.8482185932989782E-015]
i  4  component: [-1.9720682928499407E-015,1.9863586427981288E-015]

Taylor coefficients of f1(x) are included by:
Output itaylor of order 4
i  0  component: [2.4439694694070749E-307,2.4439694694070798E-307]
i  1  component: [2.4439694694070745E-307,2.4439694694070798E-307]
i  2  component: [1.2219847347035370E-307,1.2219847347035399E-307]
i  3  component: [4.0732824490117906E-308,4.0732824490117996E-308]
i  4  component: [1.0183206122529469E-308,1.0183206122529499E-308]
```

With `itaylor x(p,interval(706));` i.e. $x_1 = +706$, the program output is

```
Taylor coefficients of g(x) are included by:
Output itaylor of order 4
i  0  component: [+7.0599999999999954E+002,+7.0600000000000080E+002]
i  1  component: [+9.9999999999999811E-001,+1.0000000000000018E+000]
i  2  component: [-1.7529495936443918E-015,+1.8291647933680609E-015]
i  3  component: [-1.8482185932989782E-015,+1.8101109934371439E-015]
i  4  component: [-1.9720682928499407E-015,+1.9863586427981288E-015]

Taylor coefficients of h(x) are included by:
Output itaylor of order 4
i  0  component: [+7.0600000000000000E+002,+7.0600000000000012E+002]
i  1  component: [+9.9999999999999988E-001,+1.0000000000000000E+000]
i  2  component: [+1.2219847347035370E-307,+1.2219847347035399E-307]
i  3  component: [-4.0732824490117996E-308,-4.0732824490117906E-308]
i  4  component: [+1.0183206122529469E-308,+1.0183206122529499E-308]

Taylor coefficients of f1(x) are included by:
Output itaylor of order 4
i  0  component: [+7.0600000000000000E+002,+7.0600000000000012E+002]
i  1  component: [+9.9999999999999988E-001,+1.0000000000000000E+000]
i  2  component: [+1.2219847347035370E-307,+1.2219847347035399E-307]
i  3  component: [-4.0732824490117996E-308,-4.0732824490117906E-308]
i  4  component: [+1.0183206122529469E-308,+1.0183206122529499E-308]
```

The above results shew that $g(x)$ delivers only tight inclusions, if $x < 0$ and all the same $h(x)$, if $x > 0$. So we defined a third equivalent function $f1(x) \equiv \ln(1 + e^x)$ in such a way that the calculated inclusions of the Taylor coefficients are now sufficiently tight in all cases.

To get sufficiently tight inclusions of the Taylor coefficients of a given function f, it can be necessary to choose different expressions in different regions of the domain of f.

Remarks:

- In general the inclusions of the Taylor coefficients become more wide the higher the order of the expansion is chosen.
- Suppose, the Taylor coefficient $(f)_k$ vanishes for $x = x_0$, then in general the inclusion of the coefficient $(f)_k$ calculated for a point of expansion near x_0 can not be sufficiently tight. For example, calculate the inclusion of the first derivative of $f(x) := x/(1 + x^2)$ for $x = \texttt{succ}(1)$; $f'(1) = 0$.
- The expansion order p can be changed by the user and keeps valid until the next change is done. However, an `itaylor` object, declared with the previous order, will keep this order until an assignment operator delivers with his right-hand operand a new `itaylor` object constructed with another order.
- There is no possibility to modify the components, i.e. the values of the Taylor coefficients of an `itaylor` object outside the `itaylor` class. So further functions of type `itaylor` can only be implemented inside the class `itaylor`.
- However, with the elementary functions listed in the table of page 21, together with the four arithmetic operators, it is now possible to define a wide field of available expressions of type `itaylor`.
- Consider the four arithmetic operators and consider the case that only one operand is of type `itaylor`, then the type of the other operand must be an element of the set

  $$\{\ \texttt{int, real, interval}\ \}.$$

  Be aware that in this case these three possible operands will always be interpreted as a **constant** value and not as an independent variable.

  The result of the arithmetic operators is always of type `itaylor`, if at least one of the operands is of type `itaylor`.

2.4 Implementation, Staggered Format

In some applications it is necessary to calculate the inclusions of the Taylor coefficients with higher precision to get sufficiently tight inclusion intervals. For this purpose in C-XSC a staggered Taylor arithmetic is implemented in the modules `litaylor.hpp` and

litaylor.cpp. With the **C++** class l_itaylor the necessary tools are available for the user.

The class l_itaylor provides the following constructors:

Constructors of the class l_itaylor:

1. `l_itaylor();  // default constructor;`
2. `l_itaylor(const l_itaylor& );  // copy constructor;`
3. `l_itaylor(int order, const real& r);  //` $(r, 1, 0, ..., 0)$;
4. `l_itaylor(int order, const l_real& l);  //` $(l, 1, 0, ..., 0)$;
5. `l_itaylor(int order, const interval& z);  //` $(z, 1, 0, ..., 0)$;
6. `l_itaylor(int order, const l_interval& z);  //` $(z, 1, 0, ..., 0)$;

To avoid possible conversion errors by using the constructors 3. and 4. see the appropriate comments on page 17 and 53.

Initialization functions for independent variables of the class l_itaylor:

1. `l_itaylor var_l_itaylor(int order, const real& r);`
2. `l_itaylor var_l_itaylor(int order, const l_real& l);`
3. `l_itaylor var_l_itaylor(int order, const interval& z);`
4. `l_itaylor var_l_itaylor(int order, const l_interval& z);`

With the assignments

```
l_itaylor x;   x = var_l_itaylor(6,0.25);
```

we get the interval vector $([0.25], [1], [0], [0], [0], [0], [0])$ with $6 + 1 = 7$ components of type l_interval. This interval vector is an attribute of the object x, which can be used as an independent variable in the Taylor arithmetic.

Initialization functions for constants of the class `l_itaylor`:

1. `l_itaylor const_l_itaylor(int order, const real& r);`
2. `l_itaylor const_l_itaylor(int order, const l_real& l);`
3. `l_itaylor const_l_itaylor(int order, const interval& z);`
4. `l_itaylor const_l_itaylor(int order, const l_interval& z);`

With the assignments

```
l_itaylor c;   c = const_l_itaylor(6,0.25);
```

we get the interval vector $([0.25], [0], [0], [0], [0], [0], [0])$ with $6 + 1 = 7$ components of type `l_interval`. `c` is a now a constant value in the Taylor arithmetic.

Assignment operators of the class `l_itaylor`:

1. `l_itaylor operator = ( const l_itaylor& t );`
2. `l_itaylor operator = ( int );`
3. `l_itaylor operator = ( const real& r );`
4. `l_itaylor operator = ( const l_real& r );`
5. `l_itaylor operator = ( const interval& z );`
6. `l_itaylor operator = ( const l_interval& z );`

With the assignments

```
int p=6;   real r1, r2(3.5);   l_itaylor t(p,r1);   t = r2;
```

we get an interval vector $([3.5], [0], [0], [0], [0], [0], [0])$ with $6 + 1 = 7$ components of type `l_interval`. This interval vector is an attribute of the class object `t`. If the real value `r2` is not presentable in the IEEE system, again be aware of the described conversion errors.

Be aware that the assignment operators No. 2-6 **always** deliver a **constant** `l_itaylor` object and not an independent variable!

Access to the Taylor coefficients and derivatives; output procedure

1. `int get_order( const l_itaylor& t );`
2. `l_ivector get_all_coef( const l_itaylor& t );`
3. `l_interval get_j_coef( const l_itaylor& t, int j );`
4. `l_interval get_j_derive( const l_itaylor& t, int j );`
5. `void print_l_itaylor( const l_itaylor& t );`

Again, the easiest way to explain the above functions is a small sample program.

```
 /*-------------------------------------------------------------
   Sample program litayl_ex1.cpp for staggered Taylor arithmetic.
      Calculating guaranteed inclusions of the derivatives of the
      function  f = x / (1+x^2) in interval staggered arithmetic.
  -------------------------------------------------------------*/

#include "litaylor.hpp"   // Header file for class l_itaylor
#include <l_interval.hpp> // Interval staggered arithmetic
#include <iostream>       // Input, output

using namespace std;
using namespace taylor;
using namespace cxsc;

main()
{
    stagprec = 3;  // Provides a precision of about 3*16
                   // decimal digits.
    int p = 80;    // Taylor expansion of order p

    l_itaylor x(p,succ(1.0));  // Constructor call
                               // Point of expansion: succ(1.0)
    l_itaylor f;
    f = x / (1+sqr(x));

    cout << SetDotPrecision(16*stagprec,16*stagprec) << Scientific;
    cout << "1.  derivative: " << get_j_derive(f,1)  << endl;
    cout << "80. derivative: " << get_j_derive(f,80) << endl;
}
```

The program `litayl_ex1` produces the following output:

```
 1. Derivative:
    [-1.1102230246251561706450823457415912639734977118 64E-0016,
     -1.1102230246251561706450823457415912639734977118 51E-0016]
80. Derivative:
    [3.2546020996171047179273743366742207570020387058 67E+0106,
     3.2546020996171047179273743366742207570120057241 43E+0106]
```

Remarks:

1. Write a short program using the class `itaylor` to calculate the first derivative of the function
$$f(x) = \frac{x}{1+x^2}, \quad f'(x) = \frac{1-x^2}{(1+x^2)^2}, \quad f'(1) = 0.$$
For the argument $x = \texttt{succ}(1) = 1 + 2^{-52}$ near the zero $x_0 = 1$ of $f'(x)$ you will get the rather useless inclusion
$$f'(\texttt{succ}(1)) \in \texttt{[-1.1022302462515 65E-016,+5.5511151231257821E-017]}.$$
However, using the above sample program, already with `stagprec = 3` we get a rather tight inclusion of $f'(\texttt{succ}(1))$ with 46 correct decimal digits.

2. In general higher orders of the expansion result in lower accuracy of the calculated inclusions. However in the sample program with `p` $= 80$ we still get 38 correct decimal digits.

In the next sample program the derivatives up to order `p` $= 2$ are to be calculated in a wide region of the domain of
$$f(x) = \arctan \frac{x^2}{1+x^2}, \quad x \in \mathbb{R}.$$
Therefore $f(x)$ is implemented as follows:
$$f1(x) := \begin{cases} \arctan \dfrac{x^2}{1+x^2}, & \text{if } |x| < 1, \\ \arctan \dfrac{1}{1+(1/x)^2}, & \text{otherwise.} \end{cases}$$
In the following sample program `litayl_ex2.cpp` the inclusions of $f1(x)$ are compared with the inclusions calculated with the functions $g(x) := \arctan(x^2/(1+x^2))$ and $h(x) := \arctan(1/(1+(1/x)^2))$.

```
/*-------------------------------------------------------------------
  Sample program litayl_ex2.cpp for staggered Taylor arithmetic.
  Calculating guaranteed inclusions of the derivatives of the
  function  f = atan(x^2/(1+x^2)) in interval staggered arithmetic.
```

```
-----------------------------------------------------------------*/

#include "litaylor.hpp"   // Header file for class l_itaylor
#include <l_interval.hpp> // Interval staggered arithmetic
#include <iostream>       // Input, output

using namespace std;
using namespace taylor;
using namespace cxsc;

l_itaylor f1(const l_itaylor& x)
{
    l_itaylor w;
    l_real S( abs( Sup(get_j_derive(x,0)) ) );
    if (S<1) {
        w = sqr(x);
        w = atan( w/(1+w) );
    }
    else
        w = atan(1/(1+sqr(1/x)));  // to avoid overflow
    return w;
}

main()
{
    stagprec = 3;  // Provides a precision of about 3*16=48
                   // decimal digits.
    int p = 2;     // Taylor expansion of order p

    l_itaylor x(p,comp(0.5,-499));       // Constructor call for
                           // independentvariable x of order p=2;
    l_itaylor f,g,h;       // Default constructor call

    f = f1(x); // After assignment: f is Taylor object of order p.
    cout << SetDotPrecision(16*stagprec,16*stagprec) << Scientific
         << "Derivatives of function f1(x):" << endl;
    cout << "1. derivative: " << get_j_derive(f,1)  << endl;
    cout << "2. derivative: " << get_j_derive(f,2)  << endl;

    cout << "Derivatives of function g(x):" << endl;
    g = sqr(x); g = atan(g/(1+g)); // suitable for |(x)_0| < 1;
    cout << "1. derivative: " << get_j_derive(g,1)  << endl;
    cout << "2. derivative: " << get_j_derive(g,2)  << endl;
```

```
    cout << "Derivatives of function h(x):" << endl;
    h = atan(1/(1+sqr(1/x))); // suitable for |(x)_0| >= 1;
    cout << "1. derivative: " << get_j_derive(h,1)  << endl;
    cout << "2. derivative: " << get_j_derive(h,2)  << endl;
}
```

The above program produces with the constructor call

```
l_itaylor x(p,comp(0.5,-499))
```

the following output:

```
Derivatives of function f1(x):
1. derivative:
   [6.109872726999209364103958786427235399578805481144E-0151,
    6.109872726999209364103958786427235399578805481145E-0151]
2. derivative:
   [1.999999999999999999999999999999999999999999999934E+0000,
    2.000000000000000000000000000000000000000000000000E+0000]
Derivatives of function g(x):
1. derivative:
   [6.109872726999209364103958786427235399578805481144E-0151,
    6.109872726999209364103958786427235399578805481145E-0151]
2. derivative:
   [1.999999999999999999999999999999999999999999999934E+0000,
    2.000000000000000000000000000000000000000000000000E+0000]
Derivatives of function h(x):
Segmentation fault
```

The above program produces with the constructor call

```
l_itaylor x(p,comp(0.5,514))
```

the following output:

```
Derivatives of function f1(x):
1. derivative:
   [0.000000000000000000000000000000000000000000000000E+0000,
    9.881312916824930883531375857364427447301196052287E-0324]
2. derivative:
   [-2.964393875047479265059412757209328234190358815686E-0323,
    +9.881312916824930883531375857364427447301196052287E-0324]
Derivatives of function g(x):
Segmentation fault
```

The above program produces with the constructor call

```
l_itaylor x(p,comp(0.5,134))
```

the following output:

```
Derivatives of function f1(x):
1. derivative:
   [7.74518382969863654563606126657270369514043838392 7E-0121,
    7.74518382969863654563606126657270369514043838416 2E-0121]
2. derivative:
   [-2.13384839950567438410500487769973769303602357902 4E-0160,
    -2.13384839950567438410500487769973769303602357894 4E-0160]
Derivatives of function g(x):
1. derivative:
   [0.00000000000000000000000000000000000000000000000 0E+0000,
    7.54036756269730910175373002195558170771105241913 4E-0088]
2. derivative:
   [-2.76989358533572155428091501574718867771268823591 8E-0127,
    +6.92473396333930286063574280562029838633409043991 9E-0128]
Derivatives of function h(x):
1. derivative:
   [7.74518382969863654563606126657270369514043838392 7E-0121,
    7.74518382969863654563606126657270369514043838416 2E-0121]
2. derivative:
   [-2.13384839950567438410500487769973769303602357902 4E-0160,
    -2.13384839950567438410500487769973769303602357894 4E-0160]
```

Remarks:

1. The last constructor call `l_itaylor x(p,comp(0.5,134))` delivers an independent variable `x` of order $p = 2$ with $(x)_0 = [2^{+133}]$, which is a point interval. In contrast to $g(x)$ the functions $f1(x)$ and $h(x)$ deliver tight inclusions of the derivatives.

2. The constructor call `l_itaylor x(p,comp(0.5,-499))` delivers an independent variable `x` of order $p = 2$ with $(x)_0 = [2^{-500}]$, which again is a point interval. $f1(x)$ and $g(x)$ deliver tight inclusions of the derivatives, whereas $h(x)$ produces an error message.

3. The constructor call `l_itaylor x(p,comp(0.5,514))` delivers an independent variable `x` of order $p = 2$ with $(x)_0 = [2^{513}]$, which again is a point interval. Only the expression $\arctan[1/(1+(1/x)^2)]$ avoids an overflow and delivers an useless inclusion of the first derivative. However, the first and second derivatives are too small to be representable with several correct decimal digits in the IEEE system. Realize this by choosing `stagprec=12`.

In the last sample programs we used some of the arithmetic operators $\{+, -, *, /\}$. If at least one of the operands is an object of the class `l_itaylor`, then for the result this holds as well. If only one operand is of type `l_itaylor`, then the type of the other operand must be an element of the set

$$\{ \quad \texttt{int, real, l\_real, interval, l\_interval} \quad \}.$$

Be aware that in this case each of the five possible operands will always be interpreted as a **constant** value and not as an independent variable.

The elementary functions of type `l_itaylor`, implemented in C-XSC with the recursion formulas of page 10, see part 1-4, are listed in the following table.

Elementary Functions of the Class `l_itaylor` `l_itaylor u=u(x);  l_interval a;  int n;`			
Function	**C-XSC Name**	**Function**	**C-XSC Name**
u^2	`sqr(u)`	$\arcsin(u)$	`asin(u)`
$\sqrt{u}$	`sqrt(u)`	$\arccos(u)$	`acos(u)`
$\sqrt[n]{u}$	`sqrt(u,n)`	$\arctan(u)$	`atan(u)`
$\sqrt{1+u^2}$	`sqrt1px2(u)`	$\operatorname{arccot}(u)$	`acot(u)`
$\sqrt{1-u^2}$	`sqrt1mx2(u)`	$\sinh(u)$	`sinh(u)`
$\sqrt{u^2-1}$	`sqrtx2m1(u)`	$\cosh(u)$	`cosh(u)`
$\sqrt{1+u}-1$	`sqrtp1m1(u)`	$\tanh(u)$	`tanh(u)`
e^u	`exp(u)`	$\coth(u)$	`coth(u)`
e^u-1	`expm1(u)`	$\operatorname{arsinh}(u)$	`asinh(u)`
u^a	`pow(u,a)`	$\operatorname{arcosh}(u)$	`acosh(u)`
$\ln(u)$	`ln(u)`	$\operatorname{artanh}(u)$	`atanh(u)`
$\ln(1+u)$	`lnp1(u)`	$\operatorname{arcoth}(u)$	`acoth(u)`
$\sin(u)$	`sin(u)`		
$\cos(u)$	`cos(u)`		
$\tan(u)$	`tan(u)`		
$\cot(u)$	`cot(u)`		

The n-th root $\sqrt[n]{u}$ is defined for $(u)_0 \geq 0$ and $n \geq 1$; $\sqrt[1]{u} \equiv u$. Combining the above elementary functions with the arithmetic operators, we get expressions f of type `l_itaylor`, i.e. $(p+1)$-tuples with enclosures of the Taylor coefficients of f.

3 Functions in Two Real Variables

3.1 Introduction

Let us consider a function $f : D_f \subseteq \mathbb{R}^2 \to \mathbb{R}$ of two independent variables $x, y \in \mathbb{R}$ with continuous partial derivatives up to order p in a compact region $D_f \subseteq \mathbb{R}^2$. Under this assumption the partial derivatives are independent of the sequence of the single differentiations, and with the usual notations

$$\frac{\partial^2 f(x,y)}{\partial x\,\partial y} = f_{xy}, \qquad \frac{\partial^2 f(x,y)}{\partial y\,\partial x} = f_{yx}$$

and with $p = 3$ it holds

$$f_{xy} = f_{yx}, \qquad f_{xxy} = f_{xyx} = f_{yxx}, \qquad f_{yyx} = f_{yxy} = f_{xyy}. \tag{1}$$

The Taylor expansion can be written in the form

$$\begin{aligned} f(x_0+h, y_0+k) \;=\; & f(x_0,y_0) + \left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right) f(x_0,y_0) + \\ & + \frac{1}{2!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^2 f(x_0,y_0) + \ldots + \\ & + \frac{1}{p!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^p f(x_0,y_0) + \\ & + \frac{1}{(p+1)!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^{p+1} f(x_0+\delta h, y_0+\delta k),\ 0<\delta<1. \end{aligned}$$

For example, in the above expansion it holds

$$\frac{1}{3!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^3 f(x_0,y_0) \;=\; h^3k^0\frac{f_{xxx}}{3!0!} + h^2k^1\frac{f_{xxy}}{2!1!} + h^1k^2\frac{f_{xyy}}{1!2!} + h^0k^3\frac{f_{yyy}}{0!3!},$$

and the fractions on the right-hand side are the Taylor coefficients which will be used in the following notation:

$$f[3][0] := \frac{f_{xxx}}{3!0!}, \quad f[2][1] := \frac{f_{xxy}}{2!1!}, \quad f[1][2] := \frac{f_{xyy}}{1!2!}, \quad f[0][3] := \frac{f_{yyy}}{0!3!}.$$

The partial derivatives in the above numerators are to be evaluated for the arguments $x = x_0$ and $y = y_0$, and $f[0][0]$ is defined by $f[0][0] := f(x_0, y_0)$.

An arbitrary summand in the Taylor expansion can now be written in the form:

$$\frac{1}{n!}\left(h\frac{\partial}{\partial x} + k\frac{\partial}{\partial y}\right)^n f(x_0,y_0) = \sum_{i=0}^{n} h^{n-i}k^i \cdot f[n-i][i], \quad n = 0,1,\ldots,p. \tag{2}$$

The task is now to calculate guaranteed inclusions of the Taylor coefficients $f[n-i][i]$ with $i = 0,1,\ldots,n$ and $n = 0,1,\ldots,p$.

3.2 Implementation, IEEE Format

Let us consider the given function

$$f(x, y) = x^2 - 2xy^3.$$

To calculate guaranteed enclosures of the Taylor coefficients $f[n-i][i]$ up to order $n = p$ at the point $(x_0, y_0) = (2, -3)$ we use the following sample program

```
/*************************************************************
*  Sample program dim2tayl_ex1.cpp to calculate enclosures *
*  of the Taylor coefficients of    f(x,y)=x^2-2xy^3    at *
*  (x,y)=(2,-3) up to the desired Taylor order p=3.        *
*************************************************************/

    #include <iostream>
    #include "dim2taylor.hpp"

    using namespace std;
    using namespace cxsc;
    using namespace taylor;

    dim2taylor f(dim2taylor_vector& x)
    {   //  f = x^2 - 2*x*y^3;
        dim2taylor erg;   // Default constructor call
        erg = sqr(x[1]) - 2*x[1]*x[2]*sqr(x[2]); // f(x,y)
        return erg;
    }

    int main()
    {
        int p = 3;      // Desired maximal order of Taylor expansion

        ivector iv(2);  // 2 interval components with Lb=1 and Ub=2;
        iv[1] = interval(2);   // x-value
        iv[2] = interval(-3);  // y-value

        dim2taylor_vector tv;  // Default constructor call
        tv = init_var(p,iv);   // Initialization with vector iv
        dim2taylor t;          // Default constructor call
        t = f(tv);             // function call

        cout << "t[1][2] = " << t[1][2] << endl;
    }
```

The Sample program produces the following output:

```
t[1][2] = [ 18.000000, 18.000000].
```

Remarks:

1. The above point interval is a guaranteed inclusion of the Taylor coefficient $f[1][2] = f_{xyy}/(1! \cdot 2!) = 18$.

2. For calculating such inclusions the classes `dim2taylor` and `dim2taylor_vector` are declared and implemented in the two **C++** files

   `dim2taylor.hpp`, `dim2taylor.cpp`.

3. An object `t` of the class `dim2taylor` contains an attribute `p` for the maximal order of the Taylor expansion. The assignment `t.get_p()` delivers the order `p` of the object `t`. `int get_p()` is a member function of the class `dim2taylor`.

   The second attribute is a pointer to a dynamic block (array) of $p+1$ elements of type ivector (interval vector) including the calculated Taylor coefficients. Because of the relations in (1) the storage scheme of the coefficients is a triangle matrix. In our example, with $p = 3$, we have

   $$\left(\begin{array}{llll} f[0][0] = +112 & f[0][1] = -108 & f[0][2] = 36 & f[0][3] = -4 \\ f[1][0] = +58 & f[1][1] = -54 & f[1][2] = 18 & \\ f[2][0] = +1 & f[2][1] = 0 & & \\ f[3][0] = 0 & & & \end{array} \right)$$

   With the assignment `t[1][2]` we get an inclusion of the Taylor coefficient $f[1][2] = f_{xyy}/(1! \cdot 2!) = 18 \in$ `t[1][2]`.

4. The following constructors are implemented in the class `dim2taylor`

   - `dim2taylor();` Default constructor with `p=0`.
   - `dim2taylor(int);` The `int` parameter determines the order `p`.
   - `dim2taylor(const dim2taylor&);` Copy constructor.

5. The assignment `t.print_dim2taylor()` produces a screen output of the inclusions of the Taylor coefficients $f[n-i][i]$ with $i = 0, 1, \ldots, n$ and $n = 0, 1, \ldots, p$. `void print_dim2taylor()` is a member function of the class `dim2taylor`.

6. For an object `t` of the class `dim2taylor` an operator [ ] is defined in such a way that `t[i]` delivers the i-th. interval vector, `i` $= 0, 1, \ldots, p$. In our example `t[1]` is an interval vector of type `ivector`

   $$\texttt{t[1]} = \big(\; [+58] \quad [-54] \quad [+18] \quad [0] \; \big),$$
   $$\text{with} \quad \texttt{t[1][2]} = [+18] := [18.000000, 18.000000].$$

Be aware that `t[1][3]` $= [0]$ is not an inclusion of the Taylor coefficient $f[1][3] = -2$, because the chosen order of the Taylor expansion is $p = 3$ and so $f[1][3] = -2$ can not be calculated. The index sum $1 + 3 = 4$ must not be greater than $p = 3$.

7. To get an object of type `dim2taylor` for the independent variables x or y the following `friend` function is implemented:

```
friend dim2taylor init_var(int, int, const interval& );
```

The first parameter determines the order `p` of the Taylor expansion. The second parameter determines with the values 1 or 2 the independent variables x or y respectively, other values are not allowed. The last parameter ot type `interval` delivers the enclosure of the corresponding coordinate x_0 or y_0. (x_0, y_0) is the point of expansion. With `tx = init_var(3,1,interval(2));` we get an object `tx` of type `dim2taylor` for the independent variable x, and the assignment `tx.print_dim2taylor()` delivers the following scheme of point intervals:

$$\begin{pmatrix} [2] & [0] & [0] & [0] \\ [1] & [0] & [0] & \\ [0] & [0] & & \\ [0] & & & \end{pmatrix}, \qquad \frac{\partial x}{\partial x} = 1 \in \texttt{tx[1][0]} = [1] := [1,1].$$

With `ty = init_var(3,2,interval(-3));` we get an object `ty` of type `dim2taylor` for the variable y, and the assignment `ty.print_dim2taylor()` delivers the following scheme of point intervals:

$$\begin{pmatrix} [-3] & [1] & [0] & [0] \\ [0] & [0] & [0] & \\ [0] & [0] & & \\ [0] & & & \end{pmatrix}, \qquad \frac{\partial y}{\partial y} = 1 \in \texttt{tx[0][1]} = [1] := [1,1].$$

8. To get an object of type `dim2taylor` for the constant value $c = 0.1$ the following `friend` function is implemented:

```
friend dim2taylor init_const(int, const interval& );
```

The first parameter determines the order `p` of the Taylor expansion, and the last parameter of type `interval` delivers an enclosure of c. If we realize the interval `z` as in the program on page 51, then with `tc = init_var(3,z);` we get an object `tc` of type `dim2taylor` for $c = 0.1$, and the assignment `tc.print_dim2taylor()` delivers the following scheme of intervals:

$$\begin{pmatrix} [0.099999, 0.100001] & [0] & [0] & [0] \\ [0] & [0] & [0] & \\ [0] & [0] & & \\ [0] & & & \end{pmatrix}, \qquad \frac{\partial c}{\partial x} = \frac{\partial c}{\partial y} = \frac{\partial^2 c}{\partial x\, \partial y} = \ldots = 0.$$

9. The arithmetic operators $\{+, -, *, /\}$ are overloaded in the class `dim2taylor`. If at least one of the operands is an object of this class, then for the operator result this holds as well. If only one operand is of type `dim2taylor`, then the type of the other operand must be an element of the set

   $$\{ \quad \texttt{int, real, interval} \quad \}.$$

   Be aware that in this case each of the three possible operands will always be interpreted as a **constant** value and not as an independent variable.

10. The elementary functions of type `dim2taylor` implemented in C-XSC are listed in the following table.

Elementary Functions of the Class `dim2taylor` `dim2taylor u;  interval a;  int n;`			
Function	**C-XSC Name**	**Function**	**C-XSC Name**
u^2	`sqr(u)`	$\cot(u)$	`cot(u)`
$\sqrt{u}$	`sqrt(u)`	$\arcsin(u)$	`asin(u)`
$\sqrt{1+u^2}$	`sqrt1px2(u)`	$\arccos(u)$	`acos(u)`
$\sqrt{u^2-1}$	`sqrtx2m1(u)`	$\arctan(u)$	`atan(u)`
$\sqrt{1+u}-1$	`sqrtp1m1(u)`	$\operatorname{arccot}(u)$	`acot(u)`
e^u	`exp(u)`	$\sinh(u)$	`sinh(u)`
u^a	`pow(u,a)`	$\cosh(u)$	`cosh(u)`
u^n	`power(u,n)`	$\tanh(u)$	`tanh(u)`
$\ln(u)$	`ln(u)`	$\coth(u)$	`coth(u)`
$\ln(1+u)$	`lnp1(u)`	$\operatorname{arsinh}(u)$	`asinh(u)`
$\sin(u)$	`sin(u)`	$\operatorname{arcosh}(u)$	`acosh(u)`
$\cos(u)$	`cos(u)`	$\operatorname{artanh}(u)$	`atanh(u)`
$\tan(u)$	`tan(u)`	$\operatorname{arcoth}(u)$	`acoth(u)`

11. To calculate guaranteed inclusions of the Taylor coefficients of our example function $f(x, y) = x^2 - 2xy^3$ two objects of type `dim2taylor` must be available for the independent variables x and y and their values 2 and -3 respectively. For this purpose the class `dim2taylor_vector` is implemented. An object `tv` of this class has an attribute which is a pointer to a dynamic block (an array) of `dim` elements of type `dim2taylor`. The object `tv` contains the following attributes of type `int`:

    - `dim` is the number of elements of type `dim2taylor`.
      `tv.get_dim()` returns the value `dim` of type `int`.

- `lb` is the smallest index of the dynamic array with elements of type `dim2taylor`. `tv.get_lb()` returns the value `lb` of type `int`.
- `ub` is the greatest index of the dynamic array with elements of type `dim2taylor`. `tv.get_ub()` returns the value `ub` of type `int`. With this definitions it holds: `dim` = `ub` − `lb` + 1.
- `p_el` is the order of the Taylor expansion of the `dim` array objects of type `dim2taylor`.
  `tv.get_p_el()` returns the value `p_el` of type `int`.

12. With the default constructor call `dim2taylor_vector tv;` the following attribute values are realized: `dim` = 2, `lb` = 1, `ub` = 2, `p_el` = 1.

13. With the constructor call `dim2taylor_vector tv(11,4,8);` the following attribute values are realized: `dim` = 5, `lb` = 4, `ub` = 8, `p_el` = 11.
    So the first parameter 11 determines the order `p_el` of the Taylor expansion and the following parameters realize the values for `lb` and `ub`, which for their part determine the attribute `dim` by `dim` = 8 − 4 + 1 = 5.
    With the next copy constructor call `dim2taylor_vector tvc(tv);` a second object `tvc` is declared, which is identical with the object `tv`, however both objects allocate different storage areas.

14. For an object `tv` of the class `dim2taylor_vector` an operator [ ] is defined in such a way that `tv[i]` delivers the i-th. object in the array with elements of type `dim2taylor`, i = `lb`, ..., `ub`. As we consider a two-dimensional Taylor arithmetic, it holds `dim` = 2, and in general we have `lb` = 1 and `ub` = 2. Hence, `tv[1]` and `tv[2]` are allowed operator calls, which return objects of the class `dim2taylor`.

15. To initialize an object `tv` of the class `dim2taylor_vector` with respect to the two independent variables x and y with the values 2 and −3 respectively, the following `friend` function

    ```
    friend dim2taylor_vector init_var(int order, ivector& value);
    ```

    is implemented. The first parameter `order` determines the order of the Taylor expansion, and the second parameter `value` delivers the two interval components `interval(2)` and `interval(-3)` realizing an inclusion of the point of expansion $(x_0, y_0) = (2, -3)$, see the sample program on page 34, where `iv` is the interval vector of type `ivector`. If for example the coordinate $x_0 = 0.1$ is not presentable in the IEEE system, then, instead of `interval(0.1)` or `interval(0.1,0.1)`, an interval `z` including 0.1 must be realized as it is shown in the program on page 51.

In the next example we calulate for the given function

$$f(x, y) = \sqrt{1 + (x + y)^2}$$

a guaranteed enclosure of the Taylor coefficient $f[2][1]$ at the not presentable point of expansion $(x_0, y_0) = (10^8, 2.1 \cdot 10^8)$. The following program

```
/*************************************************************
*  Sample program dim2tayl_ex2.cpp to calculate enclosures  *
*  of the Taylor coefficient     f[2][1]   of the function  *
*  f(x,y) = sqrt(1+(x+y)^2)    at (x,y) = (10^8, 2.1*10^8)  *
*************************************************************/

    #include <iostream>
    #include "dim2taylor.hpp"

    using namespace std;
    using namespace cxsc;
    using namespace taylor;

    dim2taylor f(dim2taylor_vector& x)
    {
        //  f(x,y) = sqrt( 1+sqr(x+y) );
        dim2taylor erg;    // Default constructor call
        erg = sqrt( 1 + sqr(x[1]+x[2]) ); // f(x,y)
        return erg;
    }

    dim2taylor g(dim2taylor_vector& x)
    {
        //  g(x,y) = sqrt( 1+sqr(x+y) );
        dim2taylor erg;    // Default constructor call
        erg = sqrt1px2(x[1]+x[2]); // g(x,y)
        return erg;
    }

    int main()
    {
        int p = 3;      // Desired maximal order of Taylor expansion
        char* string1 = "[1e8,1e8]";
        char* string2 = "[2.1e8,2.1e8]";
        interval z1,z2;
        string1 >> z1;   string2 >> z2;
        ivector iv(2);  // 2 interval components with Lb=1 and Ub=2;
        iv[1] = z1;     // inclusion of x-coordinate
        iv[2] = z2;     // inclusion of y-coordinate

        dim2taylor_vector tv;
        tv = init_var(p,iv);   // Initialization with vector iv

        dim2taylor t;          // Default constructor call
```

```
    cout << SetPrecision(16,15) << Scientific;
    t = g(tv);              // function call
    cout << "Taylor coefficient t[2][1] with g(x,y):" << endl;
    cout << t[2][1] << endl;

    t = f(tv);              // function call
    cout << "Taylor coefficient t[2][1] with f(x,y):" << endl;
    cout << t[2][1] << endl;
}
```

produces the output

```
Taylor coefficient t[2][1] with g(x,y):
[-1.624218615494407E-034,-1.624218615494386E-034]

Taylor coefficient t[2][1] with f(x,y):
[-6.931673410771015E-033,1.039751011615653E-032]
```

In the above program we implemented the two functions $f(x, y)$ and $g(x, y)$ with equivalent expressions. With $f(x, y)$ we evaluate the expression in a naive manner, which leads to a rather rough enclosure. However using the implemented function `sqrtp1x2(...)` in `g(dim2taylor_vector& x)` we get a tight inclusion of the Taylor coefficient

$$f[2][1] := \frac{f_{xxy}(x_0, y_0)}{2! \cdot 1!} = -1.624\ldots \cdot 10^{-34}.$$

But don't be anxious, if you fail by searching for the optimal expression you will get sufficiently tight enclosures by using a higher precision with the implemented staggered Taylor arithmetic, see the following section.

3.3 Implementation, Staggered Format

In the last section we pointed out that in some cases a higher precision might be necessary to get sufficiently tight enclosures of the Taylor coefficients. In C-XSC a staggered Taylor arithmetic is implemented in the modules `ldim2taylor.hpp` and `ldim2taylor.cpp`. With the **C++** classes `ldim2taylor` and `ldim2taylor_vector` all necessary tools are available for the user.

An object `t` of the class `ldim2taylor` contains an attribute `p` for the maximal order of the Taylor expansion. The assignment `t.get_p()` delivers the order `p` of the object `t`. `int get_p()` is a member function of the class `ldim2taylor`.

The second attribute is a pointer to a dynamic block (array) of $p + 1$ elements of type `l_ivector` (interval vector of the staggered format) including the calculated Taylor coefficients. Because of the relations in (1) on page 33 the storage scheme of the coefficients is a triangle matrix, see page 35.

The elementary functions of type `ldim2taylor` implemented in C-XSC are listed in the following table.

Elementary Functions of the Class `ldim2taylor` `ldim2taylor u;  l_interval a;  int n;`			
Function	**C-XSC Name**	**Function**	**C-XSC Name**
u^2	`sqr(u)`	$\cot(u)$	`cot(u)`
$\sqrt{u}$	`sqrt(u)`	$\arcsin(u)$	`asin(u)`
$\sqrt{1+u^2}$	`sqrt1px2(u)`	$\arccos(u)$	`acos(u)`
$\sqrt{u^2-1}$	`sqrtx2m1(u)`	$\arctan(u)$	`atan(u)`
$\sqrt{1+u}-1$	`sqrtp1m1(u)`	$\operatorname{arccot}(u)$	`acot(u)`
e^u	`exp(u)`	$\sinh(u)$	`sinh(u)`
u^a	`pow(u,a)`	$\cosh(u)$	`cosh(u)`
u^n	`power(u,n)`	$\tanh(u)$	`tanh(u)`
$\ln(u)$	`ln(u)`	$\coth(u)$	`coth(u)`
$\ln(1+u)$	`lnp1(u)`	$\operatorname{arsinh}(u)$	`asinh(u)`
$\sin(u)$	`sin(u)`	$\operatorname{arcosh}(u)$	`acosh(u)`
$\cos(u)$	`cos(u)`	$\operatorname{artanh}(u)$	`atanh(u)`
$\tan(u)$	`tan(u)`	$\operatorname{arcoth}(u)$	`acoth(u)`

The structure of the classes `ldim2taylor` and `ldim2taylor_vector` is almost identical with the structure of the classes `dim2taylor` and `dim2taylor_vector` already described on page 35−38. Only the types `ivector` and `interval` are to be replaced by the types `l_ivector` and `l_interval` respectively.

The arithmetic operators $\{+, -, *, /\}$ are overloaded in the class `ldim2taylor`. If at least one of the operands is an object of this class, then for the operator result this holds as well. If only one operand is of type `ldim2taylor`, then the type of the other operand must be an element of the set

$$\{\ \texttt{int, real, l\_real, interval, l\_interval}\ \}.$$

Be aware that in this case each of the five possible operands will always be interpreted as a **constant** value and not as an independent variable. If one of the operands is of type `interval`, then this interval should always be a point interval, because a multiplication with a thick interval of type `interval` will reduce the accuracy to at most 16 decimal digits in contrast to the desired accuracy of `stagprec` $\cdot$ 16 digits.

For the given function

$$f(x,y) := \sqrt{1+\frac{x^2}{y}}, \qquad \text{with point of expansion:}\ \ (x_0,y_0) = (4,2),$$

the following program `ldim2tayl_ex1.cpp` calculates a guaranteed enclosure of the Taylor coefficient

$$f[2][1] := \frac{f_{xxy}(x_0, y_0)}{2! \cdot 1!} = 1.543209\ldots \cdot 10^{-3}.$$

With `stagprec` $= 3$ we choose a precision of about $3 \cdot 16 = 48$ decimal digits.

```
/*************************************************************
*  Sample program ldim2tayl_ex1.cpp to calculate an enclosure  *
*  of the Taylor coefficient      f[2][1]     of the function  *
*  f(x,y) = sqrt( 1+x^2/y)  at (x,y)=(4,2) (Staggered Arithm.) *
*************************************************************/

    #include <iostream>
    #include "ldim2taylor.hpp"

    using namespace std;
    using namespace cxsc;
    using namespace taylor;

    ldim2taylor f(ldim2taylor_vector& x)
    {   //  f = sqrt( 1+x^2/y);
        ldim2taylor erg;
        erg = sqrt(l_real(1.0) + sqr(x[1])/x[2] ); // f(x,y)
        return erg;
    }

    int main()
    {
        int p = 3;     // Maximal order of Taylor expansion
        stagprec = 3;  // Desired precision of 3*16=48 digits

        l_ivector iv(2);  // 2 components with Lb=1 and Ub=2;
        iv[1] = l_interval(4);  // x-value,   x_0 = 4;
        iv[2] = l_interval(2);  // y-value,   y_0 = 2;

        ldim2taylor_vector tv;  // Default constructor call
        tv = init_var(p,iv);    // Initialization with vector iv
        ldim2taylor t;          // Default constructor call
        t = f(tv);              // function call
        cout << SetDotPrecision(16*stagprec,16*stagprec-3)
             << Scientific;
        cout << "t[2][1] = " << t[2][1] << endl;
    }
```

The above program produces the following output

```
t[2][1] = [1.54320987654320987654320987654320987654320987 6E-0003,
           1.54320987654320987654320987654320987654320987 7E-0003]
```

which is a guaranteed enclosure of the Taylor coefficient `t[2][1]` $= f[2][1]$ with 45 correct decimal digits.

Remarks:

1. If a point of expansion is not presentable in the binary data format, see the appropriate hints on page 53.

2. Write a program to compute an enclosure of the Taylor coefficient $f[2][1]$ for the function

$$f(x, y) := \sqrt{1 + (x + y)^2}$$

with the not presentable point of expansion $(x_0, y_0) = (10^8, 2.1 \cdot 10^8)$. Use the two possible notations of $f(x, y)$ as described in the program on page 38 and compare the quality of the enclosures. A solution is given with the program on the following page.

```
/*************************************************************
*  Sample program ldim2tayl_ex2.cpp to calculate an enclosure  *
*  of the Taylor coefficient      f[2][1]     of the function  *
*  f(x,y) = sqrt( 1+(x+y)^2)    at    (x,y) = (10^8,2.1*10^8)  *
*************************************************************/

    #include <iostream>
    #include "ldim2taylor.hpp"

    using namespace std;
    using namespace cxsc;
    using namespace taylor;

    ldim2taylor f(ldim2taylor_vector& x)
    {   //  f = sqrt( 1+(x+y)^2 );
        ldim2taylor erg;
        erg = sqrt( 1+sqr(x[1]+x[2]) );
        return erg;
    }

    ldim2taylor g(ldim2taylor_vector& x)
    {   //  g = sqrt( 1+(x+y)^2 );
        ldim2taylor erg;
        erg = sqrt1px2( x[1]+x[2] );
        return erg;
    }
```

```
int main()
{
    int p = 3;     // Maximal order of Taylor expansion
    stagprec = 3;  // Desired precision of 3*16=48 digits

    char* string1 = "[1e8,1e8]";
    char* string2 = "[2.1e8,2.1e8]";

    l_interval z1,z2;
    string1 >> z1;   string2 >> z2;

    l_ivector iv(2);  // 2 components with Lb=1 and Ub=2;
    iv[1] = z1;  // x-value,   x_0 = 4;
    iv[2] = z2;  // y-value,   y_0 = 2;

    ldim2taylor_vector tv;  // Default constructor call
    tv = init_var(p,iv);    // Initialization with vector iv
    ldim2taylor t;          // Default constructor call
    t = f(tv);              // function call
    cout << SetDotPrecision(16*stagprec,16*stagprec-3)
         << Scientific;
    cout << "Function f(x,y): t[2][1] = " << t[2][1] << endl;
    t = g(tv);
    cout << "Function g(x,y): t[2][1] = " << t[2][1] << endl;
}
```

The above program produces the following output

```
Function f(x,y):
t[2][1] = [-1.624218615494395862116993292786934325760572663E-0034,
           -1.624218615494395862116993292785604946276166290E-0034]
Function g(x,y):
t[2][1] = [-1.624218615494395862116993292786129506636281943E-0034,
           -1.624218615494395862116993292786129506636281942E-0034]
```

Again using the function `sqrtp1x2(...)` in the notation of the implemented function

```
ldim2taylor g(ldim2taylor_vector& x);
```

we get a rather tight inclusion of the Taylor coefficient `t[2][1]` with 45 correct decimal digits. Compare the above results with the inclusions on page 40 calculated with the simple IEEE format.

4 Functions in One Complex Variable

4.1 Introduction

Due to the fact that the derivation rules for real and complex expressions are totally identical we can adopt the recursion formulas of the table on page 10 for calculating the Taylor coefficients of complex functions. So the implementation of the class `citaylor` can easily be done by using the class `itaylor`, where only simple changes of names are necessary. Now, of course the complete set of elementary complex functions implemented in the class `cimath` can be integrated in `citaylor`.

4.2 Implementation, IEEE Format

In C-XSC the `citaylor` arithmetic is declared in the module `citaylor.hpp` and implemented in `citaylor.cpp`. The **C++** class `citaylor` provides the following constructors:

Constructors of the class `citaylor`:

1. `citaylor();` // default constructor;
2. `citaylor(const citaylor& );` // copy constructor;
3. `citaylor(int order, const real& r);` // $(r, 1, 0, ..., 0)$;
4. `citaylor(int order, const complex& c);` // $(c, 1, 0, ..., 0)$;
5. `citaylor(int order, const interval& z);` // $(z, 1, 0, ..., 0)$;
6. `citaylor(int order, const cinterval& z);` // $(z, 1, 0, ..., 0)$;

To avoid conversion errors by using the constructors 3. and 4. see the hints on page 52. The following initialization functions for constants are implemented:

Initialization functions for constants of the class `citaylor`:

1. `citaylor const_citaylor(int order, const real& r);`
2. `citaylor const_citaylor(int order, const complex& c);`
3. `citaylor const_citaylor(int order, const interval& z);`
4. `citaylor const_citaylor(int order, const cinterval& z);`

The following initialization functions for independent variables are implemented:

Initialization functions for constant values of the class `citaylor`:

1. `citaylor var_citaylor(int order, const real& r);`
2. `citaylor var_citaylor(int order, const complex& c);`
3. `citaylor var_citaylor(int order, const interval& z);`
4. `citaylor var_citaylor(int order, const cinterval& z);`

With the assignments

```
citaylor z;   z = var_itaylor(6,complex(-2,3));
```

we get the complex interval vector

$$(([-2],[3]),([1],[0]),([0],[0]),([0],[0]),([0],[0]),([0],[0]),([0],[0]))$$

with $6+1=7$ complex interval components. $([-2],[3])$ denotes a point interval including the complex number $c = -2+3\cdot i$. `z` is now an independent variable in the Taylor arithmetic. The above complex interval vector is an attribute of the class object `z`.

To generate an independent variable z of order 6 including the not presentable complex number $c = 0.1+3.7\cdot i$ we first have to generate an inclusion interval `ci` of type `cinterval` as described on page 52. After that the following assignments are necessary:

```
citaylor z;     z = var_itaylor(6,ci);
```

Assignment operators of the class `citaylor`:

1. `citaylor operator = ( const citaylor& t );`
2. `citaylor operator = ( int );`
3. `citaylor operator = ( const real& r );`
4. `citaylor operator = ( const complex& c );`
5. `citaylor operator = ( const interval& z );`
6. `citaylor operator = ( const cinterval& z );`

With the assignments

```
int p=3;   complex c(-2,3),c1;   citaylor t(p,c1);   t = c;
```

we get an interval vector $(([-2],[3]),([0],[0]),([0],[0]),([0],[0]))$ with $3+1=4$ complex interval components, which is an attribute of the class object `t`. If the complex number $\mathtt{c} = 0.1 + 3.7 \cdot i$ is not presentable in the IEEE system, again keep in mind the described conversion errors and first realize the inclusion interval $\mathtt{ci} \ni \mathtt{c}$ as described on page 52. After that use the following assignments:

```
int p=3;   complex c;   citaylor t(p,c);   cinterval z(ci);   t = z;
```

Now `t` is a constant `citaylor` object, where the Taylor coefficient $(\mathtt{t})_0$ of order 0 is a thick complex interval `z` beeing an optimal inclusion of the not presentable complex number $0.1 + 3.7 \cdot i \in \mathtt{z}$.

Be aware that the assignment operators No. 2-6 **always** deliver a **constant** `citaylor` object and not an independent variable!

Access to the Taylor coefficients and derivatives; output procedure

1. `int get_order( const citaylor& t );`
2. `civector get_all_coef( const citaylor& t );`
3. `cinterval get_j_coef( const citaylor& t, int j );`
4. `cinterval get_j_derive( const citaylor& t, int j );`
5. `void print_citaylor( const citaylor& t );`

The easiest way to explain the above functions is a small sample program.

```
/*------------------------------------------------------------
   Sample program citayl_ex1.cpp.
   Calculating Taylor coefficients and derivatives up to
   order p=10. Expansion point x.
------------------------------------------------------------*/

#include "citaylor.hpp"  // Header file of class citaylor
#include <cinterval.hpp> // Complex interval arithmetic
#include <iostream>      // Input

using namespace cxsc;
using namespace std;
using namespace taylor;
```

```
main()
{
    int p=10;    // order of expansion
    citaylor f;  // Default constructor for f(x) = e^(-x^2).
    cinterval c; // To include the point of expansion.
    cout << "Including the point of expansion z = x + i*y" << endl;
    cout << "c = ([x,x],[y,y]) = ? "; cin >> c;
    citaylor x(p,c);      // Constructor call

    f = exp(-sqr(x));     // function f(x)

    cout << SetPrecision(15,15) << Scientific << endl;
    print_citaylor(f); // Output of Taylor coefficients

    civector derivative(0,p); // 11 components, indices: 0,1,...,10;
    for(int i=0;i<=p; i++) derivative[i] = get_j_derive(f,i);
    cout << "f(x): Derivatives up to order 10 " << endl;
    for(int i=0; i<=p; i++) // Output derivatives
        cout  << i <<"-th. derivative: " << derivative[i] << endl;
} // main
```

With the input

```
 Including the point of expansion c = x + i*y
 c = ([x,x],[y,y]) = ? ([0,0],[-0.1,-0.1])
```

the sample program `citayl_ex1` produces the following output:

```
Output citaylor of order 10
i  0  component:
   ([1.010050167084165E+000,1.010050167084170E+000],
    [0.000000000000000E+000,-0.000000000000000E+000])
i  1  component:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [2.020100334168331E-001,2.020100334168339E-001])
i  2  component:
   ([-1.030251170425853E+000,-1.030251170425849E+000],
    [-0.000000000000000E+000,0.000000000000000E+000])
i  3  component:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [-2.033567669729462E-001,-2.033567669729453E-001])
i  4  component:
   ([5.252934235615717E-001,5.252934235615737E-001],
    [-0.000000000000000E+000,0.000000000000000E+000])
i  5  component:
   ([-0.000000000000000E+000,0.000000000000000E+000],
```

```
    [1.023544437316409E-001,1.023544437316415E-001])
i  6  component:
   ([-1.785096226449127E-001,-1.785096226449119E-001],
    [-0.000000000000000E+000,0.000000000000000E+000])
i  7  component:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [-3.434440171318077E-002,-3.434440171318060E-002])
i  8  component:
   ([4.548601570405749E-002,4.548601570405768E-002],
    [-0.000000000000000E+000,0.000000000000000E+000])
i  9  component:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [8.642889618574744E-003,8.642889618574788E-003])
i  10  component:
   ([-9.270060933183031E-003,-9.270060933182992E-003],
    [-0.000000000000000E+000,0.000000000000000E+000])

f(x): Derivatives up to order 10
0-th. derivative:
   ([1.010050167084165E+000,1.010050167084170E+000],
    [0.000000000000000E+000,-0.000000000000000E+000])
1-th. derivative:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [2.020100334168331E-001,2.020100334168339E-001])
2-th. derivative:
   ([-2.060502340851706E+000,-2.060502340851698E+000],
    [-0.000000000000000E+000,0.000000000000000E+000])
3-th. derivative:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [-1.220140601837677E+000,-1.220140601837671E+000])
4-th. derivative:
   ([1.260704216547772E+001,1.260704216547777E+001],
    [-0.000000000000000E+000,0.000000000000000E+000])
5-th. derivative:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [1.228253324779691E+001,1.228253324779697E+001])
6-th. derivative:
   ([-1.285269283043371E+002,-1.285269283043365E+002],
    [-0.000000000000000E+000,0.000000000000000E+000])
7-th. derivative:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [-1.730957846344311E+002,-1.730957846344302E+002])
8-th. derivative:
   ([1.833996153187597E+003,1.833996153187606E+003],
```

```
    [-0.000000000000000E+000,0.000000000000000E+000])
9-th. derivative:
   ([-0.000000000000000E+000,0.000000000000000E+000],
    [3.136331784788403E+003,3.136331784788420E+003])
10-th. derivative:
   ([-3.363919711433459E+004,-3.363919711433444E+004],
    [-0.000000000000000E+000,0.000000000000000E+000])
```

Remarks:

1. The input `([0,0],[-0.1,-0.1])` delivers a complex interval `c` including the not presentable complex number $z = 0 - 0.1 \cdot i$. So `c` is not a point interval but a thick interval being the **optimal** inclusion of z.

2. The above complex intervals of the program output are inclusions of the Taylor coefficients and derivatives for all the points of expansion $z \in$ `c`. For example it holds $z_1 = 0 - (0.1 + 10^{-20}) \cdot i \in$ `c`, and so the intervals of the program output are also inclusions of the Taylor coefficients and derivatives with respect to the point of expansion z_1.

3. If the chosen complex interval `c` is too thick then of course the calculated inclusion intervals for the Taylor coefficients become useless.

4. It should be noted again that the complex output intervals are **guaranteed** enclosures of the **exact** Taylor coefficients of the function f in the program.

The four arithmetic operators $\{+, -, *, /\}$ are implemented in the class `citaylor`. If at least one of the operands is an object of this class, then for the operator result this holds as well. If only one operand is of type `citaylor`, then the type of the other operand must be an element of the set

$$\{\ \texttt{int, real, complex, interval, cinterval}\ \}.$$

Be aware that in this case each of the five possible operands will always be interpreted as a **constant** value and not as an independent variable.

The elementary functions of type `citaylor`, implemented in C-XSC with the recursion formulas of page 10, see part 1-4, are listed in the following table. The n-th root $\sqrt[n]{u}$ is defined for $(u)_0 \geq 0$ and $n \geq 1$; $\sqrt[1]{u} \equiv u$. Combining the elementary functions with the arithmetic operators $\{+, -, *, /\}$, we get expressions f of type `citaylor`. Such a class object f possesses among others an attribute, which is a $(p+1)$-tuple with enclosures of the Taylor coefficients of f up to order p.

Elementary Functions of the Class `citaylor` `citaylor u=u(z);  interval a;  cinterval b;  int n;`			
Function	**C-XSC Name**	**Function**	**C-XSC Name**
u^2	`sqr(u)`	$\arcsin(u)$	`asin(u)`
$\sqrt{u}$	`sqrt(u)`	$\arccos(u)$	`acos(u)`
$\sqrt[n]{u}$	`sqrt(u,n)`	$\arctan(u)$	`atan(u)`
$\sqrt{1+u^2}$	`sqrt1px2(u)`	$\operatorname{arccot}(u)$	`acot(u)`
$\sqrt{1-u^2}$	`sqrt1mx2(u)`	$\sinh(u)$	`sinh(u)`
$\sqrt{u^2-1}$	`sqrtx2m1(u)`	$\cosh(u)$	`cosh(u)`
e^u	`exp(u)`	$\tanh(u)$	`tanh(u)`
e^u-1	`expm1(u)`	$\coth(u)$	`coth(u)`
u^a	`pow(u,a)`	$\operatorname{arsinh}(u)$	`asinh(u)`
u^b	`pow(u,b)`	$\operatorname{arcosh}(u)$	`acosh(u)`
u^n	`power(u,n)`	$\operatorname{artanh}(u)$	`atanh(u)`
$\ln(u)$	`ln(u)`	$\operatorname{arcoth}(u)$	`acoth(u)`
$\sin(u)$	`sin(u)`		
$\cos(u)$	`cos(u)`		
$\tan(u)$	`tan(u)`		
$\cot(u)$	`cot(u)`		

The function `power(u,n)` is defined for $n = 0, \pm 1, \pm 2, \ldots$. If $n < 0$ the requirement $0 \notin$ `c` must be fulfilled, whereby `c` is the given complex interval including the points of expansion.

A Avoidance of Conversion Errors

A.1 Including Arbitrary Real Numbers, IEEE Format

Unfortunately most of the decimal numbers like 0.1 or 0.73 are not presentable in the IEEE System. To realize an interval `z` including the value 0.1 see the following sample program

```
/*************************************************************
*  Sample program  Taylor_apd1.cpp  to realize an optimal  *
*  inclusion z of the not presentable decimal value  0.1;  *
```

```
*  z is an interval with a minimal diameter > 0.           *
*************************************************************/

    #include <iostream>      // Input, Output
    #include <interval.hpp>  // Interval arithmetic

    using namespace std;
    using namespace cxsc;

    int main()
    {
        interval z;
        char* string = "[0.1,0.1]";
        string >> z; // z: optimal inclusion of 0.1;

        if ( diam(z) == 0 )
        { cout << "diam(z) == 0" << endl; }
        else { cout << "diam(z) > 0" << endl; }
        cout << "z = " << z << endl;

        interval z1, z2;
        z1 = interval(0.1);  z2 = interval(0.1,0.1);
        // z1 = z2 are point intervals not including 0.1 !
    }
```

The above program produces the following output

```
diam(z) > 0
z = [ 0.099999, 0.100001 ]
```

Please notice that `z` is an optimal inclusion of the decimal value 0.1 and be aware that the two constructor calls

```
z1 = interval(0.1);      z2 = interval(0.1,0.1);
```

deliver the same point intervals `z1` = `z2`, which indeed are no inclusions of the decimal value 1/10.

A.2 Including Arbitrary Complex Numbers, IEEE Format

To realize a complex interval `ci` including the not presentable complex number $c = 0.1 + 3.7 \cdot i$ consider the following sample program

```
/*************************************************************
*  Sample program  Taylor_apd2.cpp  to realize an optimal  *
*  inclusion ci of the not presentable complex number      *
*  c = 0.1 + 3.7*i;   ci is not a complex point interval!  *
```

```
**********************************************************/

    #include <iostream>       // Input, Output
    #include <cinterval.hpp>  // Complex interval arithmetic

    using namespace std;
    using namespace cxsc;

    int main()
    {
        cinterval ci;
        char* string = "([0.1,0.1],[3.7,3.7])";
        string >> ci; // ci: optimal inclusion of 0.1+3.7*i;

        if ( diam(Re(ci)) == 0 )
            cout << "diam(Re(ci)) == 0" << endl;
        else  cout << "diam(Re(ci)) > 0" << endl;
        cout << "ci = " << ci << endl;

        cinterval ci1, ci2;
        ci1 = cinterval( interval(0.1),interval(3.7) );
        ci2 = cinterval( interval(0.1,0.1),interval(3.7,3.7) );
        // ci1=ci2 are point intervals not including 0.1+3.7*i!
    }
```

The above program produces the following output

```
diam(Re(ci)) > 0
ci = ([ 0.099999, 0.100001],[ 3.699999, 3.700001])
```

Please notice that `ci` is an optimal inclusion of the complex value $c = 0.1 + 3.7 \cdot i$ and be aware that the two constructor calls

```
ci1 = cinterval( interval(0.1),interval(3.7) );
ci2 = cinterval( interval(0.1,0.1),interval(3.7,3.7) );
```

deliver the same point intervals `ci1` = `ci2`, which indeed are no inclusions of the complex value c.

A.3 Including Arbitrary Real Numbers, Staggered Format

To realize an interval `li` including the value 0.1 in the staggered correction format see the following sample program

```
/************************************************************
*  Sample program   Taylor_apd3.cpp  to realize an optimal  *
```

```
*  inclusion li in staggered format of the not presentable  *
*  real number r = 0.1;          s is not a point interval!  *
************************************************************/

    #include <iostream>        // Input, Output
    #include <l_interval.hpp>  // Staggered interval arithmetic

    using namespace std;
    using namespace cxsc;

    int main()
    {
        stagprec = 3; // Precision of about 3*16=48 decim. digits
        l_interval li;

        char* string = "[0.1,0.1]";
        string >> li;   // li: Inclusion of 0.1 in high precision

        cout << SetDotPrecision(16*stagprec,16*stagprec-3)
             << Scientific << "li = " << li <<endl;

        if (diam(li) > 0) cout << "diam(li) > 0" << endl;
        else cout << "diam(li) == 0" << endl;
    }
```

With `stagprec = 3;` we choose a precision of about $3 \cdot 16 = 48$ decimal digits. The above program produces the following output

```
li = [9.99999999999999999999999999999999999999999999999E-0002,
      1.00000000000000000000000000000000000000000000001E-0001]
diam(li) > 0
```

Another possibility to realize the same inclusion interval is to perform an interval division by `li = l_interval(1)/10`. Be aware that the decimal number 0.1 can not exactly be stored in a binary data format of any desired precision. So an interval including 0.1 will always have a positive diameter.

References

[1] Abramowitz M. and Stegun I. *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables.* National Bureau of Standards, Washington, 1964.

[2] Adams E., Kulisch U. (Eds.): *Scientific Computing with Automatic Result Verification.* I. Language and Programming Support for Verified Scientific Computation, II. Enclosure Methods and Algorithms with Automatic Result Verification, III. Applications in the Engineering Sciences. Academic Press, San Diego, 1993.

[3] Alefeld G., Grigorieff R. D. *Fundamentals of Numerical Computation (Computer-Oriented Numerical Analysis).* Computing Supplementum **2**, Springer-Verlag, Wien / New York, 1980.

[4] Alefeld G., Herzberger J. *Introduction to Interval Computations.* Academic Press, New York, 1983.

[5] Bantle A., Krämer W. *Automatic Forward Error Analysis for Floating Point Algorithms.* Reliable Computing, Vol. 7, No. 4, pp. 321-340, 2001.

[6] Berz M.,Bischoff Ch., Corliss G., Griewank A. In Berz M.,Bischoff Ch., Corliss G., Griewank A., editors, *Computational Differentiation: Techniques, Applications, and Tools,* Philadelphia, Penn, SIAM, 1996.

[7] Bräuer, M., Krämer, W. *Rückwärtsmethode zur automatischen Berechnung von worst-case Fehlerschranken.* Bericht 03/1999 aus dem Forschungsschwerpunkt "Computerarithmetik, Intervallrechnung und Numerische Algorithmen mit Ergebnisverifikation (CAVN)" an der Universität Karlsruhe, 1999.

[8] Bräuer, M.; Hofschuster, W.; Krämer, W. *Steigungsarithmetiken in C-XSC.* Preprint BUGHW-WRSWT 2001/3, Universität Wuppertal, 2001.

[9] Braune K., Krämer W. *High Accuracy Standard Functions for Real and Complex Intervals.* In Kaucher E., Kulisch U. and Ullrich Ch., editors, Computerarithmetic: Scientific Computation and Programming Languages, pages 81-114. Teubner, Stuttgart, 1987.

[10] Braune K. *Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy.* Computing Supplementum, 6:159-184, 1988.

[11] Corliss G. F. *Applications of Differentiation Arithmetic.* In [34], pp 127-148, 1988.

[12] Corliss G. F. *Automatic Differentiation Bibliography.* In [18], pp 331-353, 1991.

[13] Fischer H. C. *Schnelle automatische Differentiation, Einschliessungsmethoden und Anwendungen.* Dissertation, Universität Karlsruhe, 1990.

[14] Fischer H. C. *Range Computation and Applications.* In [43], pp 197-211, 1990.

[15] Fischer H. C. *Effiziente Berechnung von Ableitungswerten, Gradienten und Taylorkoeffizienten.* In Chatterji S. D., Fuchssteiner B., Kulisch U., Liedl R., Purkert W. (Eds.): *Jahrbuch Überblicke Mathematik 1992,* Vieweg, Braunschweig, 1992.

[16] Fischer H. C. *Automatic Differentiation and Applications.* In [2], pp 105-142, 1993.

[17] Griewank A. *On Automatic Differentiation.* In Iri M., Tanabe K. (Eds.): *Mathematical Programming: Recent Develoments and Applications,* pp 83-108, Kluwer Academic Publishers, 1989.

[18] Griewank A., Corliss G. *Automatic Differentiation of Algorithms: Theory, Implementation and Applications.* Proceedings of Workshop on Automatic Differentiation at Breckenrigde, SIAM, Philadelphia, 1991.

[19] Hammer R., Hocks M., Kulisch U., Ratz D. *C++ Toolbox for Verified Computing.* Springer-Verlag, Berlin, 1994.

[20] Higham N. J. *Accuracy and Stability of Numerical Algorithms*, 2nd ed., SIAM, 2002.

[21] Hofschuster W., Krämer W. *Ein rechnergestützter Fehlerkalkül mit Anwendungen auf ein genaues Tabellenverfahren.* Preprint Nr. 96/5, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Karlsruhe, 1996.

[22] Hofschuster W., Krämer W. *A Computer Oriented Approach to Get Sharp Reliable Error Bounds*, Reliable Computing 3, pp. 239-248, 1997.

[23] Werner Hofschuster and Walter Krämer. *FI_LIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format.* Preprint 98/7 des Instituts für Wissenschaftliches Rechnen und Mathematische Modellbildung (IWRMM) an der Universität Karlsruhe, 1998.

[24] Hofschuster W. *Zur Berechnung von Funktionswerteinschließungen bei speziellen Funktionen der mathematischen Physik.* Dissertation, Universität Karlsruhe, 2000.

[25] Iri M. *Simultaneuos Computation of Functions, Partial Derivatives and Estimates for Rounding Errors - Complexity and Practicality.* Lapan Journal of Applied Mathematics **1**, pp 223-252, 1984.

[26] Jerell M. *Automatic Differentiation and Interval Arithmetic for Estimation of disequilibrium models.* Computational Economics, 10(3):295-316, 1997.

[27] Kaucher E., Kulisch U., Ullrich Ch. *Computerarithmetic: Scientific Computation and Programming Languages.* B. G. Teubner Verlag, Stuttgart, 1987.

[28] Klatte R., Kulisch U., Lawo Ch., Rauch M. and Wiethoff A. *C-XSC: A C++ class library for extended scientific computing.* Springer, Berlin, 1993.

[29] Krämer W. *Inverse Standard Functions for Real and Complex Point and Interval Arguments with Dynamic Accuracy.* Computing Supplementum, 6:185-212, 1988.

[30] Krämer W. *A priori Worst Case Error Bounds for Floating-Point Computations*, IEEE Transactions on Computers, Vol. 47, No. 7, 1998.

[31] Kulisch U. and Miranker W. L. *Computer Arithmetic in Theory and Practice.* Academic Press, New York, 1981.

[32] Kulisch U., Miranker W. L. *A New Approach to Scientific Computation.* Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N.Y.,1982. Academic Press, New York, 1983.

[33] Moore R. E. *Interval Analysis*. Prentice Hall, Englewood Cliffs, N.J., 1966.

[34] Moore R. E. *Reliability in Computing: The Role of Interval Methods in Scientific Computing*. Proceedings of the Conference At Columbus, Ohio, September 8-11, 1987; Perspectives in Computing **19**, Academic Press, San Diego, 1988.

[35] Rall L.B. *Applications of Software for Automatic Differentiation in Numerical Computation*. In [3], pp 141-156, 1980.

[36] Rall L.B. *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science, No. textbf120, Springer-Verlag, Berlin, 1981.

[37] Rall L.B. *Differentiation and Generation of Taylor Coefficients in* PASCAL-SC. In [32], pp 291-309, 1983.

[38] Rall L.B. *Differentiation in PASCAL-SC: Type GRADIENT*. ACM TOMS **10**, pp 161-184, 1984.

[39] Rall L.B. *Optimal Implementation of Differentiation Arithmetic*. In [27], pp 287-295, 1987.

[40] Rall L.B. *Differentiation Arithmetics*. In [43], pp 73-90, 1990.

[41] Rall L.B. and Corliss G. *An introduction to automatic differentiation*. In Berz M.,Bischoff Ch., Corliss G., Griewank A., editors, *Computational Differentiation: Techniques, Applications, and Tools*, Philadelphia, Penn, SIAM, 1996.

[42] Ratschek H. and Rokne J. *Computer Methods for the Range of Functions*. Ellis Horwood Limited, Chichester, 1984.

[43] Ullrich Ch. *Computer Arithmetic and Self-Validating Numerical Methods*. (Proceedings of SCAN-89, invited papers). Academic Press, San Diego, 1990.

[44] Wolff v. Gudenberg. *Einbettung allgemeiner Rechnerarithmetik in PASCAL mittels eines Operatorkonzeptes und Implementierung der Standardfunktionen mit optimaler Genauigkeit*. Dissertation, Universität Karlsruhe, 1980.

[45] XSC website on programming languages for scientific computing with validation. `http://www.xcs.de` [October 2005].