IMM
DEPARTMENT OF MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Lyngby – Denmark

J. No. 1997-x5-94
April 15, 1997
OS

TADIFF, A FLEXIBLE C++ PACKAGE FOR AUTOMATIC DIFFERENTIATION

using Taylor series expansion

Claus Bendtsen
Ole Stauning

TECHNICAL REPORT

IMM-REP-1997-07

IMM

Contents

1 Introduction

It is well known that an analytic function $f(t)$, $f \in C^{\infty}(D, \mathbb{R})$ and $D \subset \mathbb{R}$, can be written as a Taylor series expansion in some open ball U around the point of expansion t_0

$$f(t) = \sum_{k=0}^{\infty} \frac{f^{(k)}(t_0)}{k!}(t - t_0)^k, \qquad \text{for } t \in U. \tag{1}$$

This nice property is seldom used explicitly in applications since the derivatives $f^{(k)}$ often is regarded as difficult to obtain. This is however not true – using automatic differentiation, derivatives can be obtained reasonably cheap and without too much work. In automatic differentiation the derivatives are found without any symbolic formula manipulations. Nevertheless the result is just as accurate since the actual operations used to compute the values for the derivatives are the same as in an analytic expression for the true derivatives. This also imply that the use of interval arithmetics for the computation of the derivatives will lead to an enclosure of the true result.

Tools already exists for performing automatic differentiation [Jue91, Mic91, GJU93, BS96, BRM97]. Some of these tools work as precompilers which preprocess programs (usually in FORTRAN) to include the computation of derivatives. Other tools use operator overloading (usually in C++, ADA or PASCAL-SC) to include computation of the derivatives along with the arithmetic computations in a program. Furthermore one usually distinguish between two types of computing derivatives, namely forward- and backward automatic differentiation. The forward method is mainly used for differentiating programs which has a few input variables but many function values, ie. of the type $f : C^1(\mathbb{R}^n, \mathbb{R}^m)$, $n \leq m$, whereas the backward method is superior on programs which computes a few function values but has many input variables, ie. $n > m$. Usually the forward method is preferable on functions of the type $n = m$. These rules are just rules of thumb as the optimal method depends on the actual structure of the computations performed.

Since the computation of the derivatives, using the forward- or the backward methods, again can be differentiated, it is possible to generate higher order derivatives. This strategy has been used in the C++ package FADBAD [BS96]. In this package derivatives are found by overloading the basic type of computations, eg. `double`, so that computations instead are performed using one of the overloading types `Fdouble` or `Bdouble`, depending on what method one wants to use. By overloading the generated types, using the FADBAD package itself, it is possible to generate types `BBdouble`, `BFdouble`, `FBdouble`, `FFdouble`, `BBBdouble` and so forth. This way it is possible to generate up to order P derivatives in 2^p different ways depending on ones needs, just by multiple overloading.

In this report we will describe a method which specializes in computing Taylor expansions, ie. higher order derivatives with respect to one variable. This is also possible using the FADBAD package, but since this package was designed to compute one order of derivatives at a time, it is not optimal to use it for performing Taylor expansions. A package called TADIFF has been developed to implement the Taylor expansion method. This package is compatible with the FADBAD package so it is possible to eg. compute derivatives of Taylor coefficients. This flexibility opens up for a whole new range of applications. We will later see some examples of applications.

2 The theory of Taylor arithmetics

Assume that the function $f(t)$ is analytic in an open set $D \subset \mathbb{R}$. We can form the Taylor series around the point of expansion $t_0 \in D$

$$f(t) = \sum_{k=0}^{\infty} \frac{f^{(k)}(t_0)}{k!}(t-t_0)^k, \tag{2}$$

where $f^{(k)} = \frac{d^k f}{dt^k}$. For simplicity we will write the k'th Taylor coefficient of f at the point of expansion t_0 as $(f)_k$,

$$(f)_k = \frac{f^{(k)}(t_0)}{k!} \tag{3}$$

so that Eq. (2) can be written in the more compact form

$$f(t) = \sum_{k=0}^{\infty} (f)_k (t-t_0)^k. \tag{4}$$

Note that the zero order Taylor coefficient of f in t_0, by definition, is the function value at t_0, ie. $(f)_0 = f(t_0)$.

We have an important relationship between the Taylor coefficients of f and the Taylor coefficients of f',

$$(f)_{k+1} = \frac{1}{k+1}\left(\frac{1}{k!}\frac{d^k}{dt^k}\right)\frac{df}{dt}(t_0) = \frac{1}{(k+1)}(f')_k\,. \tag{5}$$

This relationship will be used extensively. Let u and v be analytic functions and $(u)_k$, $(v)_k$ their Taylor coefficients. We have the following elementary rules

$$(u+v)_k = (u)_k + (v)_k, \tag{6}$$

$$(u-v)_k = (u)_k - (v)_k, \tag{7}$$

$$(u \cdot v)_k = \sum_{i=0}^{k}(u)_i(v)_{k-i} = \sum_{i=0}^{k}(u)_{k-i}(v)_i, \tag{8}$$

$$(u/v)_k = \frac{1}{(v)_0}\left((u)_k - \sum_{j=1}^{k}(v)_j(u/v)_{k-j}\right) \quad \text{for } (v)_0 \neq 0. \tag{9}$$

The rule for division is formed by a simple rewriting of Eq. (8). let $w = u/v$, where $(v)_0 \neq 0$. Now we have

$$\begin{aligned}(u)_k &= \sum_{j=0}^{k}(v)_j(w)_{k-j} = (v)_0(w)_k + \sum_{j=1}^{k}(v)_j(w)_{k-j} \Rightarrow \\ (w)_k &= \frac{1}{(v)_0}\left((u)_k - \sum_{j=1}^{k}(v)_j(w)_{k-j}\right).\end{aligned}$$

If one of the functions u or v, in the above binary operations, is a constant, then all of the Taylor expansion formulas shown above can be simplified considerable. This can be done, since all but the zero order Taylor coefficient of a constant is zero, ie. $u(t) \equiv C \Rightarrow (u)_0 = C$ and $(u)_j = 0$ for $j = 1, \ldots$.

Because of symmetri in Eq. (8) when $u = v$, we can make a special formula for the Taylor coefficients of the square function. For $k \geq 1$ we have

$$(u^2)_k = \sum_{i=0}^{k}(u)_i \cdot (u)_{k-i} = \begin{cases} 2\displaystyle\sum_{i=0}^{(k-1)/2}(u)_i(u)_{k-i}, & k \text{ is odd}, \\ 2\displaystyle\sum_{i=0}^{(k-2)/2}(u)_i(u)_{k-i} + (u)_{k/2}^2, & k \text{ is even}. \end{cases} \tag{10}$$

Also the formula for the square root can be obtained. Let $w = \sqrt{u}$ so that $w^2 = u$, by using Eq. (10) we obtain

$$(u)_k = \begin{cases} 2(w)_0(w)_k + \displaystyle\sum_{i=1}^{(k-1)/2}(w)_i(w)_{k-i}, & k \text{ is odd}, \\ 2(w)_0(w)_k + \displaystyle\sum_{i=1}^{(k-2)/2}(w)_i(w)_{k-i} + (w)_{k/2}^2, & k \text{ is even}. \end{cases}$$

Isolating $(w)_k = (\sqrt{u})_k$ we obtain

$$(\sqrt{u})_k = \begin{cases} \frac{1}{2(\sqrt{u})_0}(u)_k - 2\displaystyle\sum_{i=1}^{(k-1)/2}(\sqrt{u})_i(\sqrt{u})_{k-i}, & k \text{ is odd}, \\ \frac{1}{2(\sqrt{u})_0}(u)_k - 2\displaystyle\sum_{i=1}^{(k-2)/2}(\sqrt{u})_i(\sqrt{u})_{k-i} + (\sqrt{u})_{k/2}^2, & k \text{ is even}. \end{cases} \tag{11}$$

The formula for $w = u^a$, where a is a constant can be derived using Eq. (5) and Eq. (8). Assume that $u(t_0) \neq 0$, since $w' = au^{a-1}u' \Leftrightarrow w'u = au^a u' = awu'$ we have

$$\sum_{j=0}^{k-1}(w')_j(u)_{k-1-j} = a\sum_{j=0}^{k-1}(w)_j(u')_{k-1-j}, \qquad \text{for } k \geq 1.$$

From Eq. (5) we have $(w')_j = (j+1)(w)_{j+1}$. Using this relation we get

$$k(w)_k(u)_0 + \sum_{j=0}^{k-1} j(w)_j(u)_{k-j} = a\sum_{j=0}^{k-1}(k-j)(w)_j(u)_{k-j}, \qquad \text{for } k \geq 1.$$

And after isolating $(w)_k = (u^a)_k$ we obtain the formula

$$(u^a)_k = \frac{1}{k(u)_0}\sum_{j=0}^{k-1}\left(a(k-j)-j\right)(u^a)_j(u)_{k-j}, \qquad \text{for } k \geq 1. \tag{12}$$

This formula can not be used when $u(t_0) = 0$. In this case we have to use another method. The formula for $w = \exp u$ can be found in a similar way. Since $w' = wu'$ we have the relation

$$(\exp u)_k = \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(w)_j(u)_{k-j}, \qquad \text{for } k \geq 1. \tag{13}$$

formulas for cos and sin can be obtained from the relations $\cos' u = -\sin u \cdot u'$ and $\sin' u = \cos u \cdot u'$

$$\begin{aligned}(\cos u)_k &= -\frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\sin u)_j(u)_{k-j}, \qquad \text{for } k \geq 1, & (14)\\ (\sin u)_k &= \frac{1}{k}\sum_{j=0}^{k-1}(k-j)(\cos u)_j(u)_{k-j}, \qquad \text{for } k \geq 1. & (15)\end{aligned}$$

These relations have to be used pair wise.

Some other elementary functions can be expanded after the following considerations. Let $w = f(u)$ be a composite function where $w' = \frac{1}{g}u'$ and $\frac{1}{g} = \frac{df}{du}$. Assume that the Taylor coefficients of g can be obtained. Now we have $w'g = u'$, so

$$k(u)_k = \sum_{j=0}^{k-1}(w')_j(g)_{k-1-j} = \sum_{j=0}^{k-1}(j+1)(w)_{j+1}(g)_{k-(j+1)} = \sum_{j=1}^{k} j(w)_j(g)_{k-j},$$

and after isolating $(w)_k$ we obtain

$$(w)_k = \frac{1}{(g)_0}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(w)_j(g)_{k-j}\right), \qquad \text{for } k \geq 1. \tag{16}$$

The following list of functions has been expanded using Eq. (16). See [Shi93] for a more complete list.

$$\begin{aligned}
(\log u)_k &= \frac{1}{(u)_0}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\log u)_j(u)_{k-j}\right), & (17)\\
(\tan u)_k &= \frac{1}{\cos^2(u)_0}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\tan u)_j(\cos^2 u)_{k-j}\right), & (18)\\
(\sin^{-1} u)_k &= \frac{1}{\sqrt{1-(u)_0^2}}\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\sin^{-1} u)_j(\sqrt{1-u^2})_{k-j}\right), & (19)\\
(\cos^{-1} u)_k &= \frac{-1}{\sqrt{1-(u)_0^2}}\left((u)_k + \frac{1}{k}\sum_{j=1}^{k-1} j(\cos^{-1} u)_j(\sqrt{1-u^2})_{k-j}\right), & (20)\\
(\tan^{-1} u)_k &= \frac{1}{1+(u)_0^2}\cdot\left((u)_k - \frac{1}{k}\sum_{j=1}^{k-1} j(\tan^{-1} u)_j(1+u^2)_{k-j}\right), & (21)
\end{aligned}$$

all for $k \geq 1$.

The usual way of using these rules are by generating a *codelist* from the expression which is to be expanded [Moo78, Sta96]. To perform Taylor expansion of the function $f(t) = \sin(2t)t$ we first represent it by the following codelist

$$\begin{aligned}
\tau_1(\tau) &= 2 \cdot t,\\
\tau_2(\tau) &= \sin(\tau_1),\\
\tau_3(\tau) &= \cos(\tau_1),\\
f(t) &= \tau_2 \cdot t,
\end{aligned} \tag{22}$$

introducing the functions $\tau_1(t), \tau_2(t)$ and $\tau_3(t)$. We will later see why τ_3 is needed. Every rational function consisting of the elementary operations can be decomposed in this way. We use this codelist to compute a function value by evaluating τ_1 then τ_2, τ_3 and finally f given a value t_0 for t. Since $(g)_0 = g(t_0)$ this evaluation will give the zero order coefficients of the Taylor series of the functions in the left hand side of Eq. (22). When these zero order coefficients has been found we can apply the Taylor expansion formula on every elementary operation

$$\begin{aligned}
(\tau_1)_k &= 2 \cdot (t)_k, \\
(\tau_2)_k &= \frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\tau_3)_j (\tau_1)_{k-j}, \\
(\tau_3)_k &= -\frac{1}{k} \sum_{j=0}^{k-1} (k-j)(\tau_2)_j (\tau_1)_{k-j}, \\
(f)_k &= \sum_{i=0}^{k} (\tau_2)_i (t)_{k-i},
\end{aligned} \tag{23}$$

and from the Taylor series of t; $(t)_0 = t_0, (t)_1 = 1, (t)_j = 0,$ for $j \geq 2$ compute Taylor coefficients for $k \geq 1$ of τ_1 then τ_2, τ_3 and finally f. All this looks quite complicated to do, the generation of the codelist for more complex expressions, than the one shown here, are quite tedious and errorprone to do by hand. Fortunately it is quite simple to generate codelists automatically and we will later see how it can be done using operator overloading in C++ without the user actually seing the codelist.

Another important application of Taylor expansion is the expansion of a function u, which is given implicitely as the solution to an ordinary differential equation (ODE) $u' = g(u, t)$, where g is assumed to be analytic [Moo78, CC82, CKD88, Loh88, Cha89, Cor91, Rih94, Sta96]. Using Eq. (5) we have

$$(u)_k = \frac{1}{k}(g(u,t))_{k-1}. \tag{24}$$

Given an initial value $u(t_0) = (u)_0$ for some value of $t_0 = (t)_0$ we can compute the value of $(u)_1 = (g)_0$ then the value of $(u)_2 = (g)_1/2$ and so forth, until we have enough Taylor coefficients. In principle we can continue infinitely. The main difference between this scheme and the previous one, where we knew an explicit formula for the function, is that we here have to compute one order of Taylor coefficients at a time. Since we have to go through the codelist for each order of coefficients we obtain, this scheme is more complicated to implement on a computer.

3 Computational graphs

The basic idea of the packages FADBAD and TADIFF is to collect information on dependencies of variables during a program execution. This means that a variable not only contains a value, but also contain information about dependencies on other variables. It is important to understand this new property of variables in order to fully understand the usage of the packages FADBAD and TADIFF. During an execution of a program the type and the state of each variable in the program normally changes; first it is allocated, then initialized and used and finally deallocated. To be more specific about this, we define some types and states of variables.

- *Temporary variables* are variables that have been used during a computation to store intermediate values and then later discarded. This includes variables introduced by the compiler to contain temporary results in evaluations of expressions. These variables are previously used variables which are not accessible by the user in the active scope of the program.

- *Active variables* are variables which are declared in the currently active scope of the program. If a new value is assigned to an active variable the old value of the active variable will be remembered by using a temporary variable.

These two types of variables obviously exclude each other. Furthermore the variables can have one of two states.

- *Dependent variables* are variables whose values are results of expressions in which variables occur. Also an assignment to another variable, makes the assigned variable dependent.

- *Independent variables* are variables which are not dependent. Ie. variables whose values has been assigned to constants or expressions in which only constants occured. Also uninitialized variables are independent.

The type and state of a variable is dependent on its place in the program and the state of the execution. Consider the function `brussel` in Program 3.1. This function is an implementation of $f : \mathbb{R}^3 \to \mathbb{R}^2$ in Eq. (25)

Program 3.1 A simple C++ program.

```
#include <math.h>

// Declare some independent variables:
double A(2.0/5),B(6.0/5),o(M_PI/4),a(.03);

void brussel(double &xp,double &yp,double x,double y,double t)
{
  double tmp(x*(B-x*y));    // Declare dependent variable tmp

  xp=A-x-tmp+a*cos(o*t);    // xp is now a dependent variable.
  yp=tmp;                   // yp is now a dependent variable.
}                           // tmp runs out of scope, it is
                            // now a temporary variable.

void main()
{
  // Declare independent variables:
  double x(.5),y(1.5),t(0),xp,yp;

  brussel(xp,yp,x,y,t);

  // The variables xp and yp are here dependent variables.
  // They are dependent on x,y and t.
}
```

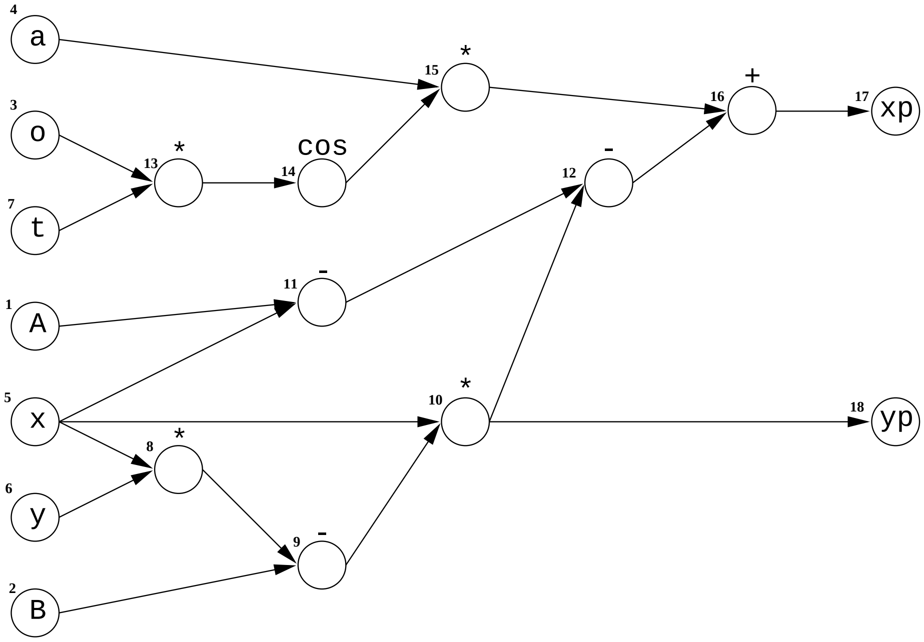

Figure 1: The DAG generated from Program 3.1.

$$f(x,y,t) = \begin{pmatrix} A + x(x \cdot y - B - 1) + a \cdot \cos(\omega \cdot t) \\ x(B - x \cdot y) \end{pmatrix} \tag{25}$$

We can also present the function as the computational graph shown in Figure 1. Each node in the graph represents a variable. A vertex corresponds to a dependency; if the arrow on a vertex is pointing to a node, then this node is dependent of the node in the other end of the vertex. A node is independent if no arrow is pointing to it. Eg. node number 5 is independent, corresponding to the independent variable `x`, node number 17 is dependent, corresponding to the variable `xp` and node number 10 is also dependent, corresponding to the temporary variable `tmp` which was used internally in `brussel`. From the graph we see that also temporary variables used internally in expression evaluations are nodes in the graph, eg. node number 8 corresponds to the subexpression `x*y` in the expression for the variable `tmp`.

Since the graph represents an actual computation, where variables only can depend on previously defined variables this graph is *acyclic*, which means that no cyclic paths can be found in the graph. The graph is therefore called a *directed acyclic graph (DAG)*.

We introduce these computational graphs because it should be more evident that the actual computation performed in a program can be transformed to a codelist. In Sec. 2 we produced the codelist Eq. (22) equivalent to the expression $f(t) = \sin(2t)t$. Introducing the variables $\tau_1, \ldots, \tau_{18}$ we can produce the following codelist equivalent to `brussel`

$$\begin{array}{rclrcl}
\tau_1 &=& \texttt{A}, & \tau_{10} &=& \tau_5 \cdot \tau_9, \\
\tau_2 &=& \texttt{B}, & \tau_{11} &=& \tau_1 - \tau_5, \\
\tau_3 &=& \texttt{o}, & \tau_{12} &=& \tau_{11} - \tau_{10}, \\
\tau_4 &=& \texttt{a}, & \tau_{13} &=& \tau_3 \cdot \tau_7, \\
\tau_5 &=& \texttt{x}, & \tau_{14} &=& \cos(\tau_{13}), \\
\tau_6 &=& \texttt{y}, & \tau_{15} &=& \tau_4 \cdot \tau_{14}, \\
\tau_7 &=& \texttt{t}, & \tau_{16} &=& \tau_{15} + \tau_{12}, \\
\tau_8 &=& \tau_5 \cdot \tau_6, & \tau_{17} &=& \tau_{16} \quad (= \texttt{xp}), \\
\tau_9 &=& \tau_2 - \tau_8, & \tau_{18} &=& \tau_{10} \quad (= \texttt{yp}),
\end{array} \tag{26}$$

where τ_i corresponds to the i'th node in Fig. 1. In general we have the following scheme, given the independent variables $x_i, \quad i = 1, \ldots m$.

$$\begin{array}{l}
\text{initialize the values:} \\
\tau_i = f_i = x_i, \quad \text{for } i = 1, \ldots m. \\
\text{compute:} \\
\textit{for } i = m+1 \text{ to } n, \\
\qquad \tau_i = f_i(\tau_1, \ldots, \tau_{i-1}),
\end{array} \tag{27}$$

where the elementary functions f_i also are allowed to be constants. Also actual computations in programs containing branching such as `if-else`, `while` and `do-while` can be written as an equivalent codelist. But it should be emphasized that these codelists does not capture information about branching in the program, but only the actual computation performed. Program 3.2 shows an example where the actual computation of a function `func` and hence its DAG- and codelist representation depends on a variable in the program. In Figure 2 and Figure 3 the two possibilities of codelists are shown.

Program 3.2 A simple C++ program with branching.

```
#include <math.h>

void func(double &out,double in)
{
  if (in>0)           // Branching occurs: The actual
    out=in*in;        // computation depends on actual
  else                // value of the input variable in.
    out=-in*in;
}

void main()
{
  // Declare independent variables:
  double fx,x;

// Case 1:
  x=-42;       // The actual computation performed in
  func(fx,x);  // func is dependent on the value of x.

// Case 2:
  x=117;       // This call is represented by a different
  func(fx,x);  // codelist from the previous function call.

}
```

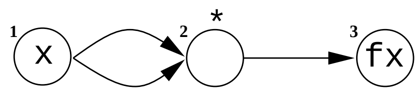

Figure 2: The DAG equivalent to `func` in Program 3.2 when `x` > 0.

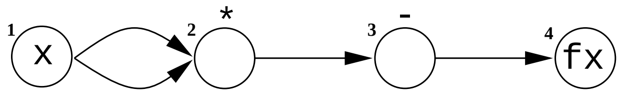

Figure 3: The DAG equivalent to `func` in Program 3.2 when `x` ≤ 0.

We have to be careful if we allow branching which is dependent on input variables, we will later use the computational graph to differentiate the program and the derivatives might be completely wrong when branching occurs. In the shown example the second order derivative of `fx` for `x=0` will only be valid for negative values. Normally we will only consider programs without branching.

4 Implementation of TADIFF

As already mentioned TADIFF is a C++ package for doing Taylor expansions of calculations in C++. TADIFF works by overloading the arithmetic operations – instead of using eg. floating point numbers for computations, the user instead changes the type of the variables from floating point type, say `double` to an overloaded type `Tdouble`[1]. Since the use of `Tdouble` instead of `double` is transparent to the program, the code will not change its existing functionality, but instead add new properties to the variables. Basically we "record" a representation of the computational graph along with the computations themselves. This graph can be accessed through the overloaded variables involved in the computations.

Assume that `x` is an independent variable while `y` is a dependent variable which is depending on `x`. Beside having the usual arithmetic operations on `x` and `y`, we also have the following three operations:

- The index operator `[]`, which is used to access the Taylor coefficients. eg. `x[i]=a` assigns the `i`'th Taylor coefficient of `x` to the value `a`. Since the values of the Taylor coefficients of dependent variables are dependent themselves, the user should only assign other values to Taylor coefficients of variables which are independent. After computing the Taylor coefficients of the dependent variables – see `y.eval(j)` later – these coefficients are accessible using the index operator, eg. `a=y[i]`.

- `y.eval(j)` computes up to order `j`'th Taylor coefficients of `y`. This operation will also compute Taylor coefficients of all the intermediate values which was used to compute `y`. Note that, since `y` is dependent on `x`, the operation will use up to order `j` Taylor coefficients of `x`. These coefficients has to be initialized by the user, either by assigning a value to `x`, before the recording of the tree, or explicitly by using the index operator `[]` as shown above. Using the index operator on a variable `x` will not disturb dependencies on `x` while assigning a constant to `x` will decouple all dependencies of `x`.

- `y.reset()` resets the Taylor coefficients of the dependent variable `y` and Taylor coefficients of all dependent variables of which `y` is dependent. The independent variables will not be affected. This operation is

[1] We here consider overloading of floating point numbers, but also overloading of other types eg. intervals, multiprecision or even another automatic differentiation type works.

necessary if one whishes to reuse the computational graph to perform several Taylor expansions.

These three additional operations can be applied on the variables after the function evaluation has finished. We will in the following see some examples on how to use these operations.

4.1 Taylor expanding a simple function

If we want to Taylor expand the function `brussel` in Program 3.1 with respect to the variable `t`. We just change the type of `t` and the variables which is dependent on `t`, in this case `tmp`, `xp` and `yp`, to the type `Tdouble`. Since `brussel` now returns a dependency graph with `xp` and `yp` as dependent variables and `t` as independent variables these variables in `main` also have to be of type `Tdouble`. Note that we do not have to change the type of `x` and `y` since these variables are independent of `t`, we will nevertheless also change the type of these variables to `Tdouble`, since this allows us the alter their values by using the index operator `[]`. We will use this later to compute more than one Taylor expansion, without having to recompute the computational graph. After the call to `brussel` we insert the code for performing the actual Taylor expansion of `xp` and `yp`. See Program 4.1 for the code which Taylor expands the function `brussel` in (`x,y,t`)=(`.5,1.5,0`). The line `t[1]=1;`, which sets the first order Taylor coefficient of `t` to the value 1, is inserted before computations of the Taylor coefficients of `xp` and `xp`. It indicates that `t` is the variable in which we want to expand the function. If the line was omitted, all Taylor coefficients, higher that order zero, will simply become zero.

Since we only use the computational graph once in Program 4.1 we do not have to reset the Taylor coefficients in the computational graph before use. If we wanted to reuse the computational graph to also expand in eg. the point (`x,y,t`)=(`2,-.5,1.8`) we could insert the following piece of code at the end of the function `main` in Program 4.1.

Program 4.1 Taylor expanding a simple C++ program.

```
#include <math.h>
#include "Tdouble.h"

// Declare some independent variables:
double A(2.0/5),B(6.0/5),o(M_PI/4),a(.03);

void brussel(Tdouble& xp,Tdouble& yp,
             Tdouble x,Tdouble y,Tdouble t)
{
  Tdouble tmp(x*(B-x*y));   // Declare dependent variable tmp

  xp=A-x-tmp+a*cos(o*t);    // xp is now dependent.
  yp=tmp;                   // yp is now dependent.
}                           // tmp runs out of scope, it is
                            // now a temporary variable.

void main()
{
  // Declare independent variables:
  Tdouble x(.5),y(1.5),t(0),xp,yp;

  brussel(xp,yp,x,y,t); // Record the computational graph.

  // The variables xp and yp are here dependent variables.
  // They are dependent on x,y and t.

  t[1]=1;                   // Taylor expansion wrt. t
  xp.eval(10);              // Compute xp[1],...,xp[10].
  yp.eval(10);              // Compute yp[1],...,yp[10].
}
```

```
xp.reset();        // Resets dependent variables of which
yp.reset();        // either xp or yp is dependent.

x[0]=2;y[0]=-.5;  // New point of expansion is inserted
t[0]=1.8;          // in the zero order coefficients.

t[1]=1;            // Taylor expansion wrt. t
xp.eval(10);       // Compute xp[0],...,xp[10].
yp.eval(10);       // Compute yp[0],...,yp[10].
```

If the two lines resetting the dependent variables `xp` and `xp` was omitted then the statements `xp.eval(10);` and `yp.eval(10);` would do nothing since the Taylor coefficients of `xp` and `xp` and all intermediate variables in the dependency graph already has been computed to order 10. It is also important to see how the values of the independent variables are changed, ie. we use `x[0]=2;` and NOT `x=2;` as the latter would decouple the variable `x` from the computational graph.

4.2 Taylor expanding the solution of an ODE

As mentioned in Sec. 2, Taylor expansion can also be used for expanding a function which is given implicitely as the solution to an ordinary differential equation (ODE) $u' = g(u,t)$. Here we have the relation from Eq. (24) in between the k'th coefficient of the solution u and the $(k-1)$'th coefficient of $g(u,t)$. Since this kind of dependency forms a kind of "feedback" in the variables which cannot be represented by a directed acyclic graph, we have to make additional code for performing this kind of Taylor expansion.

The subroutine `expand` in Program 4.2 is an example which shows how the solution of the ODE $(x', y') = f(x, y, t)$, with f given in Eq. (25), can be expanded. The `main` program starts by "recording" the computational graph for `brussel` with independent variables: `x`, `y`, `t` and dependent variables: `xp`, `yp`. These variables are then used in `expand` to access the computational graph of `brussel` for Taylor expanding the solution of the ODE, here to order 10, in the point $(x, y, t) = (.5, 1.5, 0)$.

Program 4.2 Taylor expanding the solution of an ODE.

```
#include <math.h>
#include "Tdouble.h"

double A(2.0/5),B(6.0/5),o(M_PI/4),a(.03);

void brussel(Tdouble &xp,Tdouble &yp,
             Tdouble x,Tdouble y,Tdouble t)
{
  Tdouble tmp(x*(B-x*y));   // Declare dependent variable tmp

  xp=A-x-tmp+a*cos(o*t);    // xp is now a dependent variable
  yp=tmp;                   // yp is now a dependent variable
}
void expand(Tdouble &xp,Tdouble &yp,
                Tdouble &x,Tdouble &y,Tdouble t,int order)
{
  xp.reset();              // Reset the computational
  yp.reset();              // graph of brussel.
  t[1]=1;                  // Taylor expand wrt. t.
  for(int i=0;i<order;i++) // One coefficient at a time
  {
    xp.eval(i);            // Evaluate the i'th order
    yp.eval(i);            // coefficients of xp and yp
    x[i+1]=xp[i]/(i+1);    // Use the relation:
    y[i+1]=yp[i]/(i+1);    //   (x',y')=f(x,y,t)
  }
}
void main()
{
  Tdouble x,y,t,xp,yp;     // Declare variables.

  x=.5;y=1.5;t=0;          // Specify the point of expansion.
  brussel(xp,yp,x,y,t);    // Get the computational graph of
                           // the function brussel.
  expand(xp,yp,x,y,t,10);  // Compute the Taylor Expansion of
                           // brussel in the point (.5,1.5,0)
}
```

5 Examples

In the previous section we saw how to use the TADIFF package for Taylor expanding a function which is given by an expression, and how to expand a function which is the solution of an ordinary differential equation. In this section we will see some more advanced applications using TADIFF and we will also see how to use FADBAD to differentiate the computations we perform using TADIFF.

It is important to build programs in a modular way, when applying several layers of automatic differentiation types. A good strategy for this, is to apply automatic differentiation one layer at a time. This way, programs becomes more readable and we can easily test each layer of differentiation for bugs. We will use this implementation strategy here to develop some programs which uses one or two layers of automatic differentiation.

5.1 Numerical integration

Consider the following approximation [LM62]

$$\begin{aligned}\int_a^b f(x)dx &= \frac{h}{70}\{\frac{1}{3}(41((f_a)_0+(f_b)_0)+128(f_m)_0)+h\{(f_a)_1-(f_b)_1+ \\ &\quad \frac{h}{18}\{(f_a)_2+(f_b)_2+16(f_m)_2\}\}\}+R, \qquad (28)\end{aligned}$$

where $m=(a+b)/2$, $h=b-a$ and $(f_a)_i$ is a shorthand notation for the i'th Taylor coefficient of f in the point a, etc. The error term has the form

$$R = Ch^{10}f^{(10)}(\xi), \quad \xi\in[a,b], \qquad (29)$$

$$C = \frac{1}{130977000}. \qquad (30)$$

We will use this approximation piecewise on N subintervals of the interval $[0,\pi]$ to compute the value of the integral

$$I(c,d)=\int_0^\pi \ln(c+d\cos x)dx, \qquad (31)$$

which has the true solution

$$I(c,d)=\pi\ln\left(\frac{c+\sqrt{c^2-d^2}}{2}\right). \qquad (32)$$

The following C++ function is an implementation of the integrand using the usual floating point type `double`

```
double f(double x, double c, double d)
{
  return log(c+d*cos(x));
}
```

After replacing all occurrences of `double` by `Tdouble` in the function `f` so that Taylor expansion of f is possible, we can implement the numerical integral as shown in Program 5.1. Note that in Program 5.1 the actual recording of the computational graph of `f` takes place in the first line of `I`. This line also shows a neat trick which is possible using the TADIFF package: It is possible to record the computational graph without first initializing the independent variables. This has been made possible since the computational graph can be used to produce several Taylor expansions using different values of the independent variables. One restriction to this method is, that the actual values in the graph may not be used during the recording. Program 3.2 is an example where the value of the independent variable has to be specified, before recording the graph, since the actual graph depends on this value. Since the Taylor expansion `fx.eval(2);` uses Taylor coefficients of `x` to the order 2, these coefficients has to be initialized before the expansion. This is done using the index operator `[]` on `x`. It is important to specify the Taylor coefficients of `x` to the same order as we are going to expand the function, in this case to order 2. Otherwise we will get an error message.

Since the integral $I(c,d) : \mathbb{R}^2 \to \mathbb{R}$ in Eq. (31) is a differentiable function, with the partial derivatives

$$\frac{\partial I}{\partial c}(c,d) = \frac{\pi}{\sqrt{c^2-d^2}}, \tag{33}$$

$$\frac{\partial I}{\partial d}(c,d) = -\frac{\pi \cdot d}{(c+\sqrt{c^2-d^2})\sqrt{c^2-d^2}}, \tag{34}$$

it is also possible to differentiate the program which computes the numerical approximation. To apply the backward automatic differentiation method from the FADBAD package we replace all occurrences of `double` to `Bdouble` in the functions `f` and `I`. This way `Tdouble` is modified to `TBdouble`. The function `dI` below computes the integral $I(c,d)$ and partial derivatives of $I(c,d)$ with respect to c and d.

Program 5.1 Numerical integration using Taylor Expansion.

```
double I(double c, double d, double N)
{
  Tdouble x,fx(f(x,c,d));     // Record the graph of f.
  double fa[3],fm[3],fb[3],
         h(M_PI/N),sum(0);
  int i,j;
  x[1]=1;x[2]=0;              // Specify order 1 and 2 of x.

  x[0]=0;                     // Expand f in the left point
  fx.eval(2);                 // to order 2.
  for(j=0;j<=2;j++)           // Save the 0.,1. and 2. order
    fa[j]=fx[j];              // coefficients of f.

  for(i=0;i<N;i++)
  {
    fx.reset();               // Reset before using the graph.
    x[0]=(M_PI*(2*i+1))/(2*N);// Expand f in the midpoint
    fx.eval(2);               // to order 2.
    for(j=0;j<=2;j++)         // Save the 0.,1. and 2. order
      fm[j]=fx[j];            // coefficients of f.

    fx.reset();               // Reset before using the graph.
    x[0]=(M_PI*(i+1))/N;      // Expand f in the right point.
    fx.eval(2);               // to order 2.
    for(j=0;j<=2;j++)         // Save the 0.,1. and 2. order
      fb[j]=fx[j];            // coefficients of f.

    // Compute the integral, using the Taylor Coefficients:
    sum+=(41*(fa[0]+fb[0])+128*fm[0])/3+h*(fa[1]-fb[1]+
         h*(fa[2]+fb[2]+16*fm[2])/18);

    for(j=0;j<=2;j++)     // The right endpoint is the next
      fa[j]=fb[j];        // left endpoint.
  }
  return h*sum/70;
}
```

N	$I(c,d,N)$	$\frac{\partial I}{\partial c}(5,3,N)$	$\frac{\partial I}{\partial d}(5,3,N)$
2	4.725206749584612	0.7853779605604702	−0.2617657164041858
4	4.725198468158792	0.7853982250374074	−0.2617994905324145
8	4.725198500140277	0.7853981634075534	−0.2617993878159910
16	4.725198500142803	0.7853981633974484	−0.2617993877991494
	4.725198500142803	0.7853981633974483	−0.2617993877991494

Table 1: The result of differentiating a numerical integration for an increasing number of subintervals N using the values $c = 5$ and $d = 3$. The last line shows the values obtained when evaluating the expression of $I(c,d)$ and its partial derivatives using double precision.

```
void dI(double &Ival, double &dIdc, double &dIdd,
        double c, double d, int N)
{
  Bdouble Bc(c),Bd(d),     // Initialize the input variables
          BI(I(Bc,Bd,N)); // Compute the integral.

  BI.diff(0,1);            // Compute the partial derivatives
                           // of the integral wrt. Bc and Bd.

  Ival=BI.x();             // Store the value of the integral
  dIdc=Bc.d(0);            // and its partial derivatives in
  dIdd=Bd.d(0);            // the variables Ival,dIdc and dIdd.
}
```

When calling `dI` for $c = 5$ and $d = 3$, using different values of N we obtain the numbers shown in Table 1. When we use 16 subintervals the result of the numerical integration and its partial derivatives are just as accurate as evaluating their true expressions in double precision.

5.2 Solving an initial value problem (IVP)

We have already seen in Sec. 4 how to Taylor expand the solution of an ordinary differential equation (ODE) of the form $u' = g(u,t)$ in some point $u(t_0) = u_0$. Since we only are capable of calculating a finite number of Taylor coefficients on a computer, our Taylor expansions are only local approximations to the true solution. If we wish solve the ODE for some $t \gg t_0$ we have to discretize the interval $[t_0, t]$ in some points $t_0 < t_1 < \cdots < t_n = t$ and find an approximation to the solution pointwise in $t = t_i$ for increasing values of i. By computing the Taylor polynomial of order p in the point (u_i, t_i), and evaluating this polynomial in $t = t_{i+1}$ we obtain an approximation of the solution u_{i+1} in t_{i+1},

$$u_{i+1} = \sum_{k=0}^{p} (u_i)_k (t_{i+1} - t_i)^k, \tag{35}$$

where $(u_i)_k$ denotes the k'th Taylor coefficient of the solution of $u' = g(u, t)$ in the point (u_i, t_i).

Consider the ODE $(x', y') = f(x, y, t)$ with f given in Eq. (25). The function `solve` in Program 5.2 uses the C++ implementation `brussel` of f given in Program 4.1 to compute an approximation of the solution for some $t > t_0$ based on a given initial value (x_0, y_0, t_0). The order of the Taylor expansions and the number of discretization points used in the interval $[t_0, t]$ are specified by the user. In the program we use the relation that if $w(\tau) = u(h\tau)$ we have $\frac{dw}{d\tau}(\tau) = hf(w(\tau), \tau)$. If we choose $h = t_{i+1} - t_i$ then the sum in Eq. (35) can be calculated as the sum of the Taylor coefficients for w, hence we compute the Taylor expansion for w and not u.

Since the right hand side of the ODE is a periodic function with the period $T = 2\pi/\omega$ we can consider $u(p \cdot T)$, where p is a positive integer, as a discrete map of the initial value $u(0) = u_0$. Since this map is a differentiable function we can use Newtons method for locating periodic solutions, ie. find solutions of the equation

$$u_0 - u(p \cdot T) = 0, \quad \text{and } u' = f(u, t), \;\; u(0) = u_0 \tag{36}$$

for a given positive integer value of p.

For using Newton's method we need derivatives of $u_0 - u(p \cdot T)$ with respect to the initial value u_0. Since $u(p \cdot T)$ is a function of u_0 given approximately by the C++ function `solve`, we differentiate `solve`, by replacing all occurrences of `double` with `Fdouble` given by the FADBAD package. This way we can automatically obtain derivatives of `solve` with respect to the initial values. See Program 5.3 for an implementation of Newton's method. The program will after a few iterations find the periodic $p = 2$ solution with the initial values $(x(0), y(0)) \approx (0.385047, 3.25168)$.

Program 5.2 Solving an initial value problem using Taylor expansion.

```
void solve(double &x,double &y, double &t, double tto,
           int N, int order)
{
  Tdouble Tx,Ty,Tt(0),xp,yp; // Declare variables.
  double h((tto-t)/N);
  int i,j;

  brussel(xp,yp,Tx,Ty,Tt);   // Get the computational graph of
                             // the function brussel.

  Tt[1]=h;                   // Taylor expand wrt. t.

  for(i=0;i<N;i++)
  {
    xp.reset();              // Reset the computational
    yp.reset();              // graph of brussel.

    Tx[0]=x;                 // Initialize the point of
    Ty[0]=y;                 // expansion by initializing
    Tt[0]=t;                 // the zero order coefficients.

    for(j=0;j<order;j++)     // One coefficient at a time
    {
      xp.eval(j);            // Evaluate the i'th order
      yp.eval(j);            // coefficients of xp and yp
      Tx[j+1]=xp[j]*h/(j+1); // Use the relation:
      Ty[j+1]=yp[j]*h/(j+1); //   (x',y')=f(x,y,t)

      x+=Tx[j+1];            // Evaluate the Taylor
      y+=Ty[j+1];            // polynomials.
    }
    t+=h;                    // We have a new solution point.
  }
}
```

Program 5.3 Taylor expanding the solution of an ODE, and searching for periodic solutions, using Newton's method.

```
void Newton(double &x,double &y)
{
  Fdouble Fx,Fy,Ft,IFx,IFy;
  double det,dx,dy;
  int i;

  do
  {
    Fx=x;Fy=y;Ft=0;           // Initial value of integration.

    Fx.diff(0,2);             // We want derivatives of the
    Fy.diff(1,2);             // integration wrt. x and y.

    IFx=Fx;IFy=Fy;            // Save the initial point.

    solve(IFx,IFy,Ft,16,40,7);// Solve the IVP in 2 periods.

    Fx-=IFx;Fy-=IFy;          // (Fx,Fy)=0 => periodic sol.

    det=Fx.d(0)*Fy.d(1)-Fy.d(0)*Fx.d(1);    // Compute
    dx=(Fy.d(1)*Fx.x()-Fx.d(1)*Fy.x())/det; // the Newton
    dy=(Fx.d(0)*Fy.x()-Fy.d(0)*Fx.x())/det; // correction.

    x-=dx;y-=dy;              // Make the Newton iteration.

  }while(dx*dx+dy*dy>1e-6);   // Repeat until convergence.
}

void main()
{
  double x(0.38),y(3.3);      // Initial guess.

  Newton(x,y);                // Find periodic solution.

}
```

It is worth noting that the initial value solver, when using the type `TFdouble` for evaluation, not only computes an approximation to the solution of $u' = f(u,t)$ but also computes an approximation to the solution of the variational equation $v' = D_u f(u,t)v$ where $D_u f$ denotes the Jacobian matrix of f with respect to u. All this, just by replacing `double` with `Fdouble`.

6 Conclusion

We have developed a program package for performing automatic differentiation. The package is designed to calculate several orders of derivatives of functions with respect to one variable, forming a Taylor series expansion of the function.

The program package is capable of Taylor expanding functions which is implemented as C++ programs. The program works by overloading all of the arithmetic operations in the function evaluations, and is enabled by replacing occurrences of the arithmetic type, used in the evaluations, by an overloading type, which is generated by the package. If computations are performed using the double precision type `double`, the overloading type will be called `Tdouble`. The program package is generic, so that overloading on other arithmetic types, such as intervals, also is possible.

When using the overloading types, the program will, besides computing the function value, also collect information on dependencies of variables used in the evaluation. These dependencies can be represented as a directed acyclic graph or by a codelist. Using this codelist representation we can form recursive formulas for the Taylor coefficients of all the intermediate values used in the function evaluation. After evaluating the function we apply the Taylor expansion formulas to compute Taylor coefficients of the function in the point of evaluation.

Since the Taylor expansion of a function in itself is a function evaluation, we are capable of computing derivatives of Taylor coefficients by using multiple overloading. One of the main ideas of the FADBAD and TADIFF packages is to use this aspect in a very flexible way, so that we by using the FADBAD package can compute derivatives of Taylor coefficients with respect to any variable involved in the function evaluation.

In the report we have developed a program for performing numerical integration of a function using the overloading type `Tdouble` to compute Taylor coefficients up to order 2, these Taylor coefficients are used in the numerical integration. Using the backward differentiation type `TBdouble` instead of `Tdouble` we can compute partial derivatives of the integral with respect to parameters in the integrand, without changing anything else in the code which computes the numerical integral.

Also a program for computing Taylor coefficients, to some specified order, of a function, which is a solution to an ordinary differential equation, has been developed. This Taylor expansion has been used to make a program for solving initial value problems. By differentiating this initial value problem solver, using `TFdouble` instead of `Tdouble`, we include the solution of the variational equation of the ordinary differential equation by forward automatic differentiation. These derivatives has been used in the report to locate a periodic solution of an initial value problem. Another important application, using the forward method on Taylor expansions, is validated integration of initial value problems, using interval arithmetics [CKD88, Loh88, Rih94, Sta96]. Here Taylor expansion of the variational problem is needed to obtain narrow enclosures of the solution.

Any mixture of the types Forward, Backward and Taylor are possible, using, in principle, any machine arithmetic; single precision, double precision, interval arithmetic, etc.

All this for free, download the FADBAD and the TADIFF packages and documentation from the FADBAD - TADIFF homepage

<http://www.imm.dtu.dk/fadbad.html>

References

[BRM97] C. Bischof, L. Roh, and A. Mauer. ADIC: An Extensible Automatic Differentiation Tool for ANSI-C. Technical Report ANL/MCS-P626-1196, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue Argonne, Illinois 60439, March 1997.

[BS96] Claus Bendtsen and Ole Stauning. FADBAD, A Flexible C++ Package for Automatic Differentiation. Technical Report IMM-REP-1996-17, Department of Mathematical Modelling, Technical University of Denmark, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, August 1996.

[CC82] George F. Corliss and Y.F. Chang. Solving Ordinary Differential Equations using Taylor Series. *ACM Transactions on Mathematical Software*, 8(2):114–144, 1982.

[Cha89] Y. F. Chang. Solving Stiff Systems by Taylor Series. *Appl. Math. Comput.*, 31:251–269, 1989.

[CKD88] George F. Corliss, Gary S. Krenz, and Paul H. Davis. Bibliography on interval methods for the solution of ordinary differential equations. Technical Report 289, Department of Mathematics, Statistics and Computer Science, Marquette University, Department of Mathematics, Statistics and Computer Science, Marquette University, Milwaukee, WI 53233 and School of Mathematics University of Bristol Bristol BS8 1TW, England, September 1988.

[Cor91] George F. Corliss. Automatic Differentiation Bibliography. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 331–353, Philadelphia, Penna., 1991. SIAM.

[GJU93] Andreas Griewank, David Juedes, and Jean Utke. ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++. Technical Report MCS-P180-1190, Argonne National

Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue Argonne, Illinois 60439, December 1993.

[Jue91] David W. Juedes. A Taxonomy of Automatic Differentiation Tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329, Philadelphia, Penna., 1991. SIAM.

[LM62] J. Lambert and A. R. Mitchell. On the Solution of $y' = f(x, y)$ by a Class of High Accuracy Difference Formulae of Low Order. *ZAMP*, 13:223–232, 1962.

[Loh88] Rudolf J. Lohner. *Einschließung der Lösung gewöhnlicher Anfangs- und Randwertaufgaben*. PhD thesis, Fakultät für Mathematik der Universität Karlsruhe, June 1988. In German.

[Mic91] Leo Michelotti. MXYZPTLK: A C++ Hacker's Implementation of Automatic Differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 218–227, Philadelphia, Penna., 1991. SIAM.

[Moo78] Ramon E. Moore. *Methods and Applications of Interval Analysis*. SIAM Studies in Applied Mathematics, Philadelphia, 1978.

[Rih94] Robert Rihm. Interval Methods for Initial Value Problems in ODEs. In Jürgen Herzberger, editor, *Topics in Validated Computations*, pages 173–207. IMACS–GAMM, 1994.

[Shi93] Dimitri Shiriaev. *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*. PhD thesis, Fakultät für Mathematik der Universität Karlsruhe, December 1993.

[Sta96] Ole Stauning. Enclosing Solutions of Ordinary Differential Equations. Technical Report IMM-REP-1996-18, Department of Mathematical Modelling, Technical University of Denmark, Department of Mathematical Modelling, Technical University of Denmark, 2800 Lyngby, Denmark, 1996.