IMM
DEPARTMENT OF MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Lyngby – Denmark

J. No. 1996-x5-94
August 15, 1996
OS

FADBAD, a flexible C++ package for automatic differentiation

using the forward and backward methods

Claus Bendtsen
Ole Stauning

TECHNICAL REPORT

IMM-REP-1996-17

IMM

Copyright © 1996 by Claus Bendtsen and Ole Stauning.

All rights reserved.

The FADBAD code is provided ”as is”, without any warranty of any kind, either expressed or implied, including but not limited to, any implied warranty of merchantibility or fitness for any purpose. In no event will any party who distributed the code be liable for damages or for any claim(s) by any other party, including but not limited to, any lost profits, lost monies, lost data or data rendered inaccurate, losses sustained by third parties, or any other special, incidental or consequential damages arising out of the use or inability to use the program, even if the possibility of such damages has been advised against. The entire risk as to the quality, the performance, and the fitness of the program for any particular purpose lies with the party using the code.

The FADBAD code, and any derivative of the code, may not be used in a commercial package without the prior explicit written permission of the authors. Verbatim copies of the code may be made and distributed in any medium, provided that this copyright notice is not removed or altered in any way. No fees may be charged for distribution of the codes, other than a fee to cover the cost of the media and a reasonable handling fee.

Contents

1 Introduction

The importance of differentiation as a mathematical tool is obvious. One of the first things we learn in elementary school is how to manually differentiate expressions using a few elementary formulas. Unfortunately the use of derivatives in scientific computing has been quite limited due to the misunderstanding that derivatives are hard to obtain. Many people still think that the only alternative to the symbolic way of obtaining derivatives is to use divided differences in which the difficulties in finding an expression for the derivatives are avoided. But by using divided differences, truncation errors are introduced and this usually has a negative effect on further computations – in fact it can lead to very inaccurate results.

The use of a symbolic differentiation package such as Maple or Mathematica can solve the problem of obtaining expressions for the derivatives. This method obviously avoids truncation errors but usually these packages have problems in handling large expressions and the time/space usage for computing derivatives can be enormous. In worst case it can even cause a program to crash. Furthermore, common subexpressions are usually not identified in the expressions and this leads to unnecessary computations during the evaluation of the derivatives.

Automatic differentiation is an alternative to the above methods. Here derivatives are computed by using the very well known "chain rule" for composite functions, in a clever way. In automatic differentiation the evaluation of a function and its derivatives are calculated simultaneously, using the same code and common temporary values. If the code for the evaluation is optimized, then the computation of the derivatives will automatically be optimized. The resulting differentiation is free from truncation errors, and if we calculate the derivatives using interval analysis we will obtain enclosures of the true derivatives. Automatic differentiation is *"easy"* to implement in languages with operator overloading such as C++, Ada and PASCAL-XSC, see e.g. [Jue91] for a survey of available tools.

FADBAD is a C++ program package which combines the two basic ways of applying the chain rule, namely forward- and backward automatic differentiation. Both the forward- and the backward differentiation methods use operator overloading to redefine the arithmetic operations, so that the program is capable of calculating first order derivatives. The only thing a user has to provide is the C++ program that performs the evaluation of the function. Since the computation of the derivatives is itself a C++ program we can obtain higher order derivatives by building the forward- and the backward automatic differentiation classes on top of each other. Using this approach we can obtain derivatives of order p in 2^p different ways – hereby giving the possibility to minimize time/space usage of the computations by choosing the optimal combination of the algorithms. Depending, of course, on the function which we want to differentiate.

2 The Theory of Automatic Differentiation

As mentioned above the key idea behind automatic differentiation is the "chain rule" which is used on a program or part of it in order to obtain partial derivatives of all *output* variables with respect to all *input* variables.

As will be explained in Sec. 3 any program can be considered as a sequence of n *elementary* functions, $\{f_i\}_{i=1,\dots,n}$, where f_i is a function only of the variables $\tau_1,\dots,\tau_{i-1}$ and $\tau_i = f_i(\tau_1,\dots,\tau_{i-1})$. Using a vector representation we write, $\underline{\tau} = \mathsf{f}(\underline{\tau})$, where $\underline{\tau} = \{\tau_i\}_{i=1,\dots,n}$. A matrix, $\mathbf{Df}$ containing derivatives of f with respect to $\underline{\tau}$ and a matrix, $\mathbf{D}\tau$ containing partial derivatives of $\underline{\tau}$ are defined,

$$\mathbf{Df} = \left\{\frac{\partial f_i}{\partial \tau_j}\right\}_{i,j=1,\dots,n} = \begin{pmatrix} 0 & \dots & & \\ \frac{\partial f_2}{\partial \tau_1} & 0 & \dots & \\ \frac{\partial f_3}{\partial \tau_1} & \frac{\partial f_3}{\partial \tau_2} & 0 & \dots \\ \dots & \ddots & \ddots & \ddots \end{pmatrix}, \tag{1}$$

$$\mathbf{D}\tau = \left\{\frac{\partial \tau_i}{\partial \tau_j}\right\}_{i,j=1,\dots,n} = \begin{pmatrix} 1 & 0 & \dots & \\ \frac{\partial \tau_2}{\partial \tau_1} & 1 & \ddots & \\ \frac{\partial \tau_3}{\partial \tau_1} & \frac{\partial \tau_3}{\partial \tau_2} & 1 & \ddots \\ \dots & \ddots & \ddots & \ddots \end{pmatrix}. \tag{2}$$

The chain rule for composite functions

$$\frac{\partial \tau_i}{\partial \tau_j} = \delta_{ij} + \sum_{k=j}^{i-1} \frac{\partial f_i}{\partial \tau_k}\frac{\partial \tau_k}{\partial \tau_j}, \quad \text{where} \quad \delta_{ij} = \begin{cases} 1 & i = j \\ 0 & \text{otherwise} \end{cases}$$

for $j \le i \le n$ can now be formulated as

$$\mathbf{D}\tau = \mathbf{I} + \mathbf{Df}\mathbf{D}\tau,$$

where $\mathbf{I}$ is the identity matrix.

Now $\mathbf{D}\tau$ can be isolated and we have

$$\begin{aligned} \mathbf{D}\tau = \mathbf{I} + \mathbf{Df}\mathbf{D}\tau \quad &\Leftrightarrow \\ (\mathbf{I} - \mathbf{Df})\mathbf{D}\tau = \mathbf{I} \quad &\Leftrightarrow \qquad (3) \\ \mathbf{D}\tau = (\mathbf{I} - \mathbf{Df})^{-1} \quad &\Leftrightarrow \\ \mathbf{D}\tau(\mathbf{I} - \mathbf{Df}) = \mathbf{I} \quad &\Leftrightarrow \\ (\mathbf{I} - \mathbf{Df})^T \mathbf{D}\tau^T = \mathbf{I} \quad & \qquad (4) \end{aligned}$$

where the solution of the triangular systems Eq. (3) and (4) represent the two different methods used in automatic differentiation – namely the forward and backward methods respectively.

The solution of Eq. (3) is given by forward substitution

$$
\begin{aligned}
&\textit{for } i = 2 \text{ to } n, \\
&\quad \textit{for } j = 1 \text{ to } i-1, \\
&\qquad \frac{\partial \tau_i}{\partial \tau_j} = \sum_{k=j}^{i-1} \frac{\partial f_i}{\partial \tau_k} \frac{\partial \tau_k}{\partial \tau_j}
\end{aligned}
\tag{5}
$$

and the solution of Eq. (4) is given by backward substitution

$$
\begin{aligned}
&\textit{for } j = n-1 \text{ downto } 1, \\
&\quad \textit{for } i = n \text{ downto } j+1, \\
&\qquad \frac{\partial \tau_i}{\partial \tau_j} = \sum_{k=j+1}^{i} \frac{\partial f_k}{\partial \tau_j} \frac{\partial \tau_i}{\partial \tau_k}.
\end{aligned}
\tag{6}
$$

3 Computational Graphs

In the previous section we saw how to use the chain rule in two different ways to obtain derivatives. To apply these methods in a clever way using operator overloading we have to introduce a way of interpreting a computation.

In a computer program variables are needed to save values for later use and to communicate values between different parts of the program. Usually temporary variables contain values which are only used locally in time, and erased or overwritten when the value is no longer needed. In this report, variables which are not directly accessible to the programmer but used internally during the evaluation of an expression, will also be considered as temporary variables. E.g. the calculation, `w=x*(y+z)` introduces the value of `y+z` and the value of `x*(y+z)` as temporary variables.

Furthermore we need to classify variables according to their use as follows:

- A *dependent variable* is a variable which has been assigned to another dependent variable or to an expression in which variables occured (i.e. a non-constant expression). E.g. after the assignment `w=x*(y+z)` the variable `w` is a dependent variable.

- An *independent variable* is a variable which has not been used yet, or which has been assigned to a constant, another independent variable or to an expression where no variables appeared. E.g. after `w=117*cos(13)` the variable `w` is independent.

A way of representing a given calculation is by introducing a computational graph where each node represents a variable (temporary or not temporary) and the edges represents the dependencies. Such a graph is called a Directed Acyclic Graph (DAG) because any node in the graph can only depend on previously generated nodes, [Shi93]. Consider Program 3.1 – a simple C++ program which performs the iteration $(x,y) \mapsto (\cos(x+4*y), \sin(4*x+y))$ with the initial values $x = 0.3$ and $y = 0.4$. When it finishes the iteration, it will have the dependency graph shown in Figure 1 and the variables `x,y,t1,t2` will be independent while `f,g` will be dependent, since they are functions of `x` and `y`. It is crucial that `x` and `y` are not redefined – if they were then `f` and `g` would not be functions of them any more. Notice that `t1` and `t2` are made independent deliberately by assigning them to a constant, `t1=t2=0`.

Program 3.1 A simple C++ program.

```
void test1(){
adtype x(0.3),y(0.4),t1,t2,f,g;
int p=2;

// initialize:
  f=x; g=y;

  for (int i=1;i<=p;i++) {
// loop number i:
    t1=cos(f+4*g);
    t2=sin(4*f+g);
    f=t1; g=t2;
  }

// make t1, t2 independent variables:
  t1=t2=0;

// end of test1
}
```

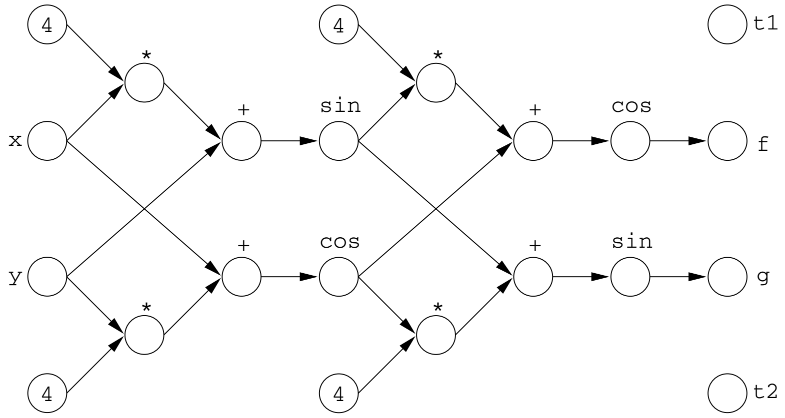

Figure 1: The DAG generated from Program 3.1.

From the above it should be evident that when a program is running on a sequential computer, it will perform a sequence of *elementary* function evaluations, $\{f_i\}_{i=1,\ldots,n}$ Each of these function evaluations will use one or more previously generated variables $\tau_1,\ldots,\tau_{i-1}$ to generate a new variable τ_i with the computed value, $f_i(\tau_1,\ldots,\tau_{i-1})$. Thus any computation can be represented by the scheme,

$$\begin{aligned}
&\text{initialize the values:}\\
&\tau_i = f_i = x_i, \quad \text{for } i = 1,\ldots m.\\
&\text{compute:}\\
&\textit{for } i = m+1 \text{ to } n,\\
&\qquad \tau_i = f_i(\tau_1,\ldots,\tau_{i-1}).
\end{aligned} \tag{7}$$

Where the elementary functions, f_i are also allowed to be constants.

In practice the arity (number of dependencies) of the elementary functions are low, usually $0-2$, which means that the matrix $\mathbf{Df}$ given by (1) is very sparse. Let a_i be the arity of the i'th function f_i and define the map

$$\kappa_i : \{1,\ldots,a_i\} \mapsto I_i \subset \{1,\ldots,i-1\},$$

so that

$$\tau_i = f_i(\tau_{\kappa_i 1},\ldots,\tau_{\kappa_i a_i}).$$

The forward substitution (5) can now be calculated as

$$\begin{aligned}
&\textit{for } i = 2 \text{ to } n,\\
&\quad \textit{for } j = 1 \text{ to } i-1,\\
&\qquad \frac{\partial \tau_i}{\partial \tau_j} = \sum_{k=1}^{a_i} \frac{\partial f_i}{\partial \tau_{\kappa_i k}} \frac{\partial \tau_{\kappa_i k}}{\partial \tau_j},
\end{aligned} \tag{8}$$

while the backward substitution (6) can be calculated as

$$\begin{aligned}
&\textit{for } j = n-1 \text{ downto } 1,\\
&\quad \textit{for } i = n \text{ downto } j+1,\\
&\qquad \frac{\partial \tau_i}{\partial \tau_j} = \sum_{k \in \mathcal{J}_j} \frac{\partial f_k}{\partial \tau_j} \frac{\partial \tau_i}{\partial \tau_k},
\end{aligned} \tag{9}$$

where $\mathcal{J}_j = \{t \mid j \in I_t\}$ denote the indices of the variables which are functions of τ_j.

Since we in general only are interested in derivatives with respect to our initial values $\{x_i\}_{i=1\dots m}$ only some of the elements in the matrix $\mathbf{D}\tau$ have to be computed.

When using the forward algorithm, the variables $\{\hat{\tau}_{i,j}\}_{i=1,\dots,n,\ j=1,\dots,m}$ are introduced and the derivatives can then be calculated alongside of the elementary functions,

$$\begin{aligned}
&\text{initialize the values:}\\
&\tau_i = x_i,\ \hat{\tau}_{ij} = \delta_{ij},\ \text{for } i, j = 1, \dots m.\\
&\text{compute:}\\
&\textit{for } i = m+1 \text{ to } n,\\
&\quad \tau_i = f_i(\tau_{\kappa_i 1}, \dots, \tau_{\kappa_i a_i}),\\
&\quad \hat{\tau}_{i,j} = \sum_{k=1}^{a_i} \frac{\partial f_i}{\partial \tau_{\kappa_i k}} \hat{\tau}_{\kappa_i k, j} \text{ for } j = 1, \dots m.
\end{aligned} \tag{10}$$

The partial derivatives are then in $\hat{\tau}$, in fact $\hat{\tau}_{i,j} = \frac{\partial \tau_i}{\partial \tau_j}$ for $i = 1, \dots, n$, $j = 1, \dots, m$.

On a computer the elementary function evaluations can easily be redefined, using operator overloading, so that both the evaluation and the calculation of the derivatives with respect to all of the independent variables are performed subsequently.

When using backward automatic differentiation one has to specify which of the dependent variables $\{\tau_i\}_{i=m+1,\ldots,n}$, one want to differentiate. Assume that $\mathcal{D}$ is the set of indices of the dependent variables that we want to differentiate. Introduce the variables $\{\hat{\tau}_{i,j}\}_{i\in\mathcal{D},\, j=1,\ldots,n}$. Now the evaluation of our elementary functions and the calculation of the derivatives can be done in one forward and one reverse sweep by

initialize the forward sweep:

$\tau_i = x_i$, for $i = 1,\ldots m$.

forward sweep (function evaluation):

for $i = m+1$ to n,

$\quad \tau_i = f_i(\tau_{\kappa_i 1},\ldots,\tau_{\kappa_i a_i})$,

initialize the reverse sweep:

$$\hat{\tau}_{i,j} = \begin{cases} 0, & i \neq j, \\ 1, & i = j, \end{cases} \quad \text{for } i \in \mathcal{D},\ j = 1,\ldots,n.$$

reverse sweep (function differentiation):

for $j = n$ downto $m+1$,

$$\hat{\tau}_{i,\kappa_j k} = \hat{\tau}_{i,\kappa_j k} + \frac{\partial f_j}{\partial \tau_{\kappa_j k}} \hat{\tau}_{i,j} \text{ for } i \in \mathcal{D},\ k = 1,\ldots,a_j. \tag{11}$$

After which we will have $\hat{\tau}_{i,j} = \frac{\partial \tau_i}{\partial \tau_j}$ for $i \in \mathcal{D},\ j = 1,\ldots,n$.

It is seen that in order to use the backward algorithm all the dependencies from the forward sweep have to be "remembered". Some of the terms $\frac{\partial f_j}{\partial \tau_{\kappa_j k}}$ in Eq. (11) use the values $\tau_{\kappa_j k}$ which are calculated during the forward sweep. One way to save all this information is to "record" the evaluations in the forward sweep by use of operator overloading. Thus, after the forward sweep one will have a representation of the computational graph and this can then, during the reverse sweep, be traversed in the opposite direction of which it was recorded.

There are some main differences between the two methods:

- If only a few dependent variables need to be differentiated with respect to a large amount of input variables then the backward algorithm is generally faster. But if a large amount of dependent variables need to be differentiated with respect to a few independent variables, the forward algorithm should normally be chosen. Of course the right choice of algorithm is completely dependent on the structure of the DAG.

- Due to the “recording” in the backward algorithm the space usage during the evaluation is linear with the time of the evaluation, and can therefore be very large for complex functions.

4 Implementation

FADBAD relies heavily on the operator overloading available in C++, for details on the topic see [C++95]. Basically the idea is to define a set of classes for the variables used in the computation one wishes to differentiate and then to use operator overloading to perform the normal computation and the differentiation on the classes as if they were just your normal variables. The result is that the code used for the normal computation and the code used for the differentiation are identical except for the variable declaration and an identification of dependent and independent variables. The details on how to use FADBAD will be described in the next section but it should be evident that the use of classes and operator overloading makes it possible to make a very user friendly package (since all the differentiation happens "behind the back" of the user) and it is our hope that we in FADBAD have achieved this goal.

As already explained the forward and backward algorithm are structurally very different. In the forward algorithm the differentiation is carried out alongside of the function evaluation and when differentiating it is convenient to store the partial derivatives in the classes of the dependent variables. For the backward algorithm the operator overloading is used to form the DAG during the forward sweep, i.e. alongside of the function evaluation. During the reverse sweep the DAG is traversed "backwards" and the class of each occuring variable stores the partial derivative with respect to itself, of the dependent variables that one wishes to differentiate. Thus, at the end of the reverse sweep the partial derivatives with respect to the independent variables are stored in the classes corresponding to the independent variables. The fact that the partial derivatives are stored in the classes of either the dependent or independent variables often makes it difficult to determine where to look for them – especially when combinations of the two methods are build on top of each other in order to obtain higher order derivatives.

Another significant difference between the forward and backward method is the use of storage. In the forward algorithm there is no need to store temporaries when they are no longer used in the function evaluation. This is however not the case for the backward method where the "recording" requires the storage of *all* temporaries until the differentiation is taking place and since most programs lead to quite a few temporaries the storage cost can be high.

The package also takes advantage of the fact that C++ allows you to view real numbers (i.e. `float` or `double` variables) as classes. The implementation has been made so that basically *any* class can be differentiated as long as the usual operators (i.e. `*`,`+`,`sin`,`exp` etc.) as well as member functions for copying and assigning the class are defined. To our knowledge this is at present the only package which has this flexibility. As will be seen in App. B this is used to differentiate intervals which can be used e.g. when doing optimization using interval analysis. And this is also the reason why higher order derivatives can so easily be obtained by just differentiating the class itself.

Since C++ is yet not an international standard different implementations have chosen to handle things differently – one noticeable difference is the use of templates. It would be natural to define the backward and forward classes as templates, but even though the current working paper on the standard, [C++95] clearly states how templates should work – the current implementations are not able to do this. The main problem is the instantiation of the templates – it is not trivial to do this correctly when the different operators introduce intermediate classes and the current version of FADBAD therefore uses macros to achieve a *template-like* functionality. The actual classes defined in the implementation are documented in App. A for the user who would like to enhance their functionality, but in most cases the next section should provide the necessary information for using FADBAD successfully.

5 User's Guide

The current version of FADBAD can be obtained from the FADBAD homepage, `http://www.imm.dtu.dk/documents/users/os/fadbad.html` or by anonymous ftp to `ftp://ftp.uni-c.dk/uni-c/unicbe/FADBAD`.

5.1 Installation

After unpacking the package the first thing to do is to run the configuration script, `configure`. Initially one is asked which C++ compiler to use and how to run the C++ preprocessor. Then the *BaseType* has to be entered. This is the class name of the type which has to be differentiated – by default the base type is `double`. If some other class is entered, one is also asked to specify an include file which defines the class. Then the number of levels in the library is requested, i.e. the maximum order of derivatives needed. Then one is asked if a production version should be created and if not then if a debugging version should be build. The difference between the different versions is the amount of error checks which are performed. The production version only checks for user errors, the non-production version also checks for internal errors (e.g. invalid pointers, incorrect resource counters) and provides a few functions for printing the DAG for the backward method. The debugging version provides a lot output showing what is going on during the function evaluation and differentiation. Finally one is asked if an extended library should be created. The extended library provides functions for easily accessing and storing the derivatives as well as defining the independent and dependent variables.

Once the `Makefile` has been created the configuration file, `config.h` has to be edited. Here one defines which operators are available on the base type. By default `config.h` is configured so that it matches the operators available on `double` or `float` but if one is e.g. using intervals as the base type it is desirable to distinguish the case `pow(interval,int)` from `pow(interval,interval)` and thus `config.h` is edited to reflect this.

Now the package can be build by invoking `make`. When the package is build it can be installed by using `make install` which will install the files in the default installation directories (if not specified by the user using `--includedir` or `--libdir` when invoking `configure`). If the extended library was build for the type `double` with at least 3 levels then it can be tested by running `make` in the directory `./test`.

5.2 Using the Extended Library

The extended library is basically an "easy-to-use" interface to the basic library. In many cases this will be the natural way to use the FADBAD package.

Initially the `.h` files should be included in the program which defines the computation one wishes to differentiate. The include files are named by the type they differentiate and the combination of the forward and backward algorithm they use. In case one wishes to calculate all 3rd order derivatives on `double` variables by one step of the backward algorithm followed by two steps of the forward algorithm, then one would include the files `BFFdouble.h` and `BFFdoublext.h`, where the latter is the include file for the extended library.

Now all the variables that are used in the function evaluation we wish to differentiate need to be declared with the included type, i.e. for the example above as `BFFdouble`. Before doing the actual computation the independent variables must be marked by using the `independent` function. The first argument is the number of independent variables and the subsequent arguments are their addresses. Now the computation can be performed. After the computation the dependent variables are marked in much the same way as the independent ones. One additional argument of type `diffquot` for storing the derivatives is needed for the `dependent` function and it is given as the first argument. Typically a subclass of `diffquot` named `diffs` is used (this only stores the needed derivatives, i.e. not both $\partial^2 f/\partial xy$ and $\partial^2 f/\partial yx$). Our example, Program 3.1 modified for use with the extended library is shown as Program 5.1. **It is of uttermost importance that all declared, dependent variables are marked – if one does not wish to differentiate some dependent variables these must be made independent explicitly.**

The partial derivatives can now be accessed from `diffs` by using the member function `get`. It has an integer argument list. The first argument is the order of the derivative and the second is the index of the dependent variable. Then the remaining arguments are the indexes of the independent variables. The function can be called with the third argument being a vector of indices of the independent variables. The indices are given by the order of the arguments for the call to the `independent` and `dependent` function – so that the first independent and dependent variable has index 0 the next index 1 and so forth.

The `independent` and `dependent` functions can also be called with a vector of pointers to the independent/dependent variables instead of the variable argument list.

One of the problems when using the extended library is that a lot of things

Program 5.1 A simple C++ program using the extended library.

```
#include "BFFdouble.h"
#include "BFFdoublext.h"

void test1(){
BFFdouble x(0.3),y(0.4),t1,t2,f,g;
int p=2;

// mark independent variables
independent(2,&x,&y);

// initialize:
  f=x; g=y;

  for (int i=1;i<=p;i++) {
// loop number i:
    t1=cos(f+4*g);
    t2=sin(4*f+g);
    f=t1; g=t2;
  }

// make t1, t2 independent variables:
  t1=t2=0;

// end of test1

// declare diffs object and mark dependent variables
diffs d;
dependent(d,2,&f,&g);

// display all 3rd order partial derivatives
// of f and g wrt. x and y.
cout<<d;

}
```

are taking place behind the user's back. The user does not have to distinguish between the forward and backward classes and does not have to worry about which variables to differentiate etc. – the cost however is a loss of flexibility. E.g. it is not possible to have nested levels of the differentiations as the differentiation is defined through **one** call of both `independent` and `dependent` and they make sure that the differentiation is carried out for all the levels in the FADBAD class hierarchy

5.3 Using the (Basic) Library

When using the basic library it is the user's responsibility to initiate the differentiation and when using higher order derivatives this has to be done separately for each level. Even though the forward and backward classes are very different we have tried to make a homogeneous interface to the two classes.

In order to initiate the differentiation one must call the member function `diff(i,n)`, where `i` is the (unique) index of the object (starting from 0) and `n` is the number of variables for which `diff` is called for the current level of differentiation. Recall that the forward method propagates the derivatives into the dependent variables during the function evaluation. Therefore `diff` is called on the independent variables one wishes to differentiate wrt., prior to the function evaluation when using the forward method. For the backward method `diff` must be called for **all** the dependent variables after the function evaluation and the derivatives are then propagated into the independent variables.

The value of an object is returned by the `x` member function and derivative number `i` is returned by calling the member function `d(i)`. In Program 5.2 and 5.3 the use of the basic library is illustrated.

When calculating higher order derivatives one has to remember that *both calculated derivatives and function values become dependent variables*. When using the backward differentiation, one therefore must either make them independent or differentiate them. In Program 5.4 an example of the nested use of the forward and backward classes is presented. The member function, `root` gives the base value of an object, i.e. it traverses all levels of forward and backward classes and returns the value of the object. This function can conveniently be used to make dependent variables independent without changing their value as shown in Program 5.4.

Clearly it is not a trivial task to decide which variables have to be differentiated or made independent as well as where to access the partial derivatives

Program 5.2 A simple C++ program using forward differentiation from the basic library.

```
#include "Fdouble.h"

void test1(){
Fdouble x(0.3),y(0.4),t1,t2,f,g;
int p=2;

// call diff on the indep. variables we wish to diff. wrt.
  x.diff(0,2); // second argument is 2 because we wish to
  y.diff(1,2); // differentiate wrt. 2 independent variables.

// initialize:
  f=x; g=y;

  for (int i=1;i<=p;i++) {
// loop number i:
    t1=cos(f+4*g);
    t2=sin(4*f+g);
    f=t1; g=t2;
  }

// make t1, t2 independent variables:
  t1=t2=0;

// end of test1
// partial derivatives has now been calculated
f.d(0); // df/dx
f.d(1); // df/dy
g.d(0); // dg/dx
g.d(1); // dg/dy
}
```

Program 5.3 A simple C++ program using backward differentiation from the basic library.

```
#include "Bdouble.h"

void test1(){
Bdouble x(0.3),y(0.4),t1,t2,f,g;
int p=2;

// initialize:
  f=x; g=y;

  for (int i=1;i<=p;i++) {
// loop number i:
    t1=cos(f+4*g);
    t2=sin(4*f+g);
    f=t1; g=t2;
  }

// make t1, t2 independent variables:
  t1=t2=0;

// end of test1
// call diff on dependent variables.
  f.diff(0,2); // second argument is 2 because we have
  g.diff(1,2); // 2 dependent variables.

// partial derivatives has now been calculated
x.d(0); // df/dx
x.d(1); // dg/dx
y.d(0); // df/dy
y.d(1); // dg/dy
}
```

when building many layers of the forward and backward classes on top of each other. In Sec. 6 an easier approach to building classes on top of each other will be shown.

Program 5.4 A "simple" C++ program using the basic library.

```
#include "BBFdouble.h"

void test1(){
BBFdouble x(0.3),y(0.4),t1,t2,f,g;
int p=2;

// call diff on forward class.
x.x().x().diff(0,2);
y.x().x().diff(1,2);

// ... function evaluation deleted ...

// call diff on backward classes.
f.diff(0,2); g.diff(1,2);
x.d(0).diff(0,4); x.d(1).diff(1,4);
y.d(0).diff(2,4); y.d(1).diff(3,4);
f.x() = f.root(); g.x() = g.root();

// now all partial derivatives are available.
x.d(0); x.d(1); y.d(0); y.d(1); // df/dx, dg/dx, df/dy, dg/dy
x.x().d(0); x.x().d(1); // d2f/dxx, d2g/dxx
x.x().d(2); x.x().d(3); // d2f/dyx, d2g/dyx
y.x().d(0); y.x().d(1); // d2f/dxy, d2g/dxy
y.x().d(2); y.x().d(3); // d2f/dyy, d2g/dyy
x.x().d(0).d(0); x.x().d(0).d(1); // d3f/dxxx, d3f/dxxy
x.x().d(1).d(0); x.x().d(1).d(1); // d3g/dxxx, d3g/dxxy
x.x().d(2).d(0); x.x().d(2).d(1); // d3f/dyxx, d3f/dyxy
x.x().d(3).d(0); x.x().d(3).d(1); // d3g/dyxx, d3g/dyxy
y.x().d(0).d(0); y.x().d(0).d(1); // d3f/dxyx, d3f/dxyy
y.x().d(1).d(0); y.x().d(1).d(1); // d3g/dxyx, d3g/dxyy
y.x().d(2).d(0); y.x().d(2).d(1); // d3f/dyyx, d3f/dyyy
y.x().d(3).d(0); y.x().d(3).d(1); // d3g/dyyx, d3g/dyyy
}
```

6 Building a Newton solver based on FADBAD

In Sec. 5 we saw that it can be difficult to determine which of the variables in a computation that are dependent and which are independent. This is especially a problem when we use several layers of differentiation classes on top of each other.

In this section we will see how to use subroutines and locally defined variables to limit the amount of active variables.

Program 6.1 A C++ subroutine.

```
#define x in[0]
#define y in[1]
#define f out[0]
#define g out[1]
int p=2;
void func(double *in,double *out){
  double t1,t2;
  f=x;g=y;
  for(int i=1;i<=p;i++){
    t1=cos(f+4*g);
    t2=sin(4*f+g);
    f=t1;g=t2;
  }
}
#undef x
#undef y
#undef f
#undef g
```

Consider Program 6.1, here the previous used example is encapsulated in a function named `func`, the function has two arrays of type `double` as arguments, each array of length 2. The argument `in` contains the independent variables for the function, while `out` is used to return the dependent variables from the evaluation. All the temporary variables used inside the function are local, so we do not have to consider if they are independent or not when we use the function.

Imagine that we in Program 6.1 replace all occurrences of the word `double` by the word `Fdouble` without changing anything else. It is now possible to calculate the Jacobian matrix of `func` by calling the member function `diff` on the independent variables `in` before the function evaluation. We use this in Program 6.2 to perform Newton iterations on `func`.

Program 6.2 A Newton iteration.

```
#include "Fdouble"

void func(Fdouble *in,Fdouble *out){
  // .. function evaluation deleted ..
}
void newton(double *val){
  Fdouble in[2],out[2];
  double det,dfdx,dfdy,dgdx,dgdy;
  do{
    in[0]=val[0];in[1]=val[1];
    in[0].diff(0,2); // call diff on the independent
    in[1].diff(1,2); // variables of func.
    func(in,out);
    dfdx=out[0].d(0);dfdy=out[0].d(1); // df/dx ; df/dy
    dgdx=out[1].d(0);dgdy=out[1].d(1); // dg/dx ; dg/dy

    det=dfdx*dgdy-dgdx*dfdy;
    val[0]-=(dgdy*out[0].x()-dfdy*out[1].x())/det;
    val[1]-=(dfdx*out[1].x()-dgdx*out[0].x())/det;

  }while(out[0]*out[0]+out[1]*out[1]>1e-6);
}
```

The function `newton` works by giving it an array of `double` in the argument `val`, when a solution is found (when the 2-norm of the function value is less than 10^{-6}) the loop is terminated and the new value of `val` is returned. Note that `val` is used as an argument as well as returning the value of `newton`. Nevertheless it is actually possible to differentiate the Newton iteration without too much work.

Assume that the occurrence of the word `double` in Program 6.2 is replaced by the word `Bdouble`, this way `double` becomes `Bdouble`, and `Fdouble` becomes `FBdouble`. Now the Newton iterations can be differentiated by calling the function `newton` as shown in Program 6.3.

Program 6.3 Differentiating a Newton iteration.

```
#include "Bdouble"
#include "FBdouble"

void func(FBdouble *in,FBdouble *out){
  // .. function evaluation deleted ..
}
void newton(Bdouble *val){
  // .. Newton iteration deleted ..
}

int main(){
  Bdouble in[2],out[2];
  in[0]=.4;in[1]=.5;         // initial values
  out[0]=in[0];out[1]=in[1]; // save the independent variables
  newton(out);
  out[0].diff(0,2); // Differentiate the dependent
  out[1].diff(1,2); // variables

  out[0].x(); // x-result of the Newton iterations
  out[1].x(); // y-result of the Newton iterations
  in[0].d(0);in[0].d(1); // derivatives wrt. in[0]
  in[1].d(0);in[1].d(1); // derivatives wrt. in[1]

  return 1;
}
```

operation	FF	FB	BF	BB
+	534	1182	642	1842
-	114	234	138	258
*	1032	1008	1080	1008
/	36	60	36	60
pow	24	0	24	0
unary -	144	24	120	0
sin	108	108	108	108
cos	108	108	108	108
TOTAL	2100	2724	2256	3384

Figure 2: The total amount of used flops in the program using different types of differentiation.

It is important to note that the call of `newton` in Program 6.3, overwrites the calling argument `out`. As already mentioned the derivatives are propagated into the independent variables when using backward differentiation, hence it is necessary to retain these independent variables. This is done in Program 6.3 by assigning another set of variables to the independent variables before the call, this way we will still have the independent variables after the call.

A problem which we will not go into detail in this report, is the problem of assuming that our programs are differentiable. In the Newton example we differentiate the `newton` function without any problems. But if we look further into the code, we will see that this assumption is wrong. The `do{..}while()` construction determines how many iterations are taken, making the result non-differentiable. Nevertheless the derivatives of `newton` are usually meaningful in a local sense. If the `do{..}while()` construction was replaced with a constant number of iterations the `newton` function would be differentiable.

It is a simple task to modify Program 6.3 to use different combinations of differentiations, in total 4 different versions can be made. It is difficult to say which version is the most optimal with respect to the number of arithmetic operations – it depends completely on the structure of the DAG which represents the computation. In Figure 2 the amount of used operations are shown, the operations has been obtained by replacing all doubles with a similar type which also counts the number of flops (floating point operations). From the figure we see that the optimal differentiation method in this example is two forward classes on top of each other.

A Overview of the Class Hierarchy

In the following we try to give an *overview* of the classes used in FADBAD so that the interested user will be able to understand and modify the code if necessary. As already mentioned in Sec. 4 the use of templates has been simulated by using macros and some of these will be described initially. Then the forward and backward classes will be described followed by the extended functions and finally we will comment on the classes for storage of derivatives.

A.1 Basic Macros and Defines – `macros.h`

The define, `Dtype` has the name of the class to be differentiated. If this is not a base class, i.e. the class to be differentiated is already one of the FADBAD classes, then the define, `BaseType` will hold the name of the base class, e.g. `double`.

The most heavily used macro is the `prefix` macro which returns the first argument added as prefix to the current `Dtype`. This is used to give each class a unique name and is useful for generating a template-like functionality.

A few other macros are used for debugging purposes and the like – their use is straight forward.

A.2 The Forward Class – `fadiff.cc`

The forward differentiation is managed by only one class. The class name is given by `prefix(adtype)` but by default adtype is redefined so that the class name becomes “F” concatenated with `Dtype`.

The class contains a variable and an array of type `Dtype` for storing the function values and derivatives respectively. The differentiation is carried out alongside the function evaluation by overloading operators and mathematical functions.

A.3 The Backward Classes – `badiff.cc`

The classes are divided in several layers. At the bottom we have the `node` class which holds the function value and derivatives in much the same way as in the forward case. There is however a need for a “resource counter” on the node so that it is possible to decide when a node is no longer needed. We then have the op class which has a pointer to a node and manages the resource counter of the node as well as the logical operators. The op class is not used directly.

On top of the op class we have the UNop and BINop classes which are used for unary and binary operators respectively. The former has one pointer to an op object the latter two – and by assigning these pointers the DAG is build during the forward sweep. These classes are not used directly.

On top of the UNop class is the adtype class which is the class the user will allocated variables for backward differentiation as. It contains the functions for initiating and accessing the derivatives and it has the only overloadings of the "=" operator.

Also on top of the UNop class are all the common unary operators, e.g. unary +, sine and square root. They each have their own class which is formed by the macros UNOPCODE, UNOPCLASS and UNOPMATCH. The first macro is just an interface to the latter two – where UNOPCLASS defines the class and UNOPMATCH defines the interface to the class so that the class will be allocated when the unary operator in question is used on the op class or subclasses thereof. Upon allocation, which happens during the forward sweep, the class just performs the calculation given by its operator, but during the reverse sweep the member function propagateop performs the differentiation.

Of special concern is the deallocation of intermediate results – temporaries. The draft for the C++ standard, [C++95, Subsec. 12.2.3] clearly states that temporaries should be deallocated "as the last step in evaluating the full-expression that contains the point where they were created" and based on this an allocated class corresponding to a unary operation will often be deallocated shortly after its allocation. In order to be able to record the DAG one thus needs to copy the class prior to the deallocation – this is done from the adtype class for the assignment operator as well as in the constructor. Many C++ implementations do however not conform with the proposed standard and deallocates temporaries at a later time (if this is the case then DELAYTEMPDEALLOC is defined by the configure script). In order for this not to be a problem we have introduced the "shadow class" defined through the macro UNOPCLASSTMP which simply acts as an empty shell around the class of the unary operator.

For the binary operators the macros BINOPCODE, BINOPCLASS and BINOPMATCH define classes on top of the BINop class in much the same way as for the unary operators.

The classes for the unary and binary operations always creates nodes to hold the calculated value and the derivatives. The adtype class does however often just link to a node created by one of these classes and can therefore be viewed as encapsulating the class of the basic calculations. This encapsulation which can take place in arbitrarily many layers – each corresponding to a reference to the result stored in the node – is reflected by the resource counter in the `node` class. Thus when performing the reverse sweep the resource counter is decremented as derivatives are propagated into the node and as the last derivative has been propagated the derivatives of the node can be propagated onwards – up through the DAG. The same mechanism is used to deallocated the DAG – when the last refering object for a node is deallocated the node itself is deallocated and the references it used are removed, possibly leading to further deallocations.

A.4 The Extended Functions – `fadiffxt.cc` and `badiffxt.cc`

The aim of the extended functions is that the user should not worry about how to initiate the differentiation correctly as well as where to get the derivatives from, no matter which combination of forward and backward classes is used on top of each other.

The call of the `independent` function stores a reference to all the independent variables in a global array and when calling the `dependent` function all the levels of forward and backward classes are traversed and the reverse sweeps (if any) are performed. Basically this function automatizes the procedure described in Subsec. 5.3. The derivatives are stored by using the `diffquot` class.

A.5 Storage of the Derivatives – `adiff_tools.cc`

When storing derivatives the symmetry implies that it is not necessary to store all partial derivatives (e.g. $\partial^2 f/\partial xy$ is equal to $\partial^2 f/\partial yx$). The classes `diffquot` and `diffs` are used for storing only the “needed” derivatives.

The class `diffquot` just calculates offsets etc. but does not perform any storage of derivatives – thus a user can easily subclass this if control of the storage is needed. To insert derivatives, the member function `insert` is called with the index of the differentiated dependent variable, the level of differentiation and indexes of the independent variables which the differentiation has occured with respect to. The member function `calcoffs` can then be used to calculate the offset corresponding the partial derivative taking the symmetry of the derivatives into account. This class only defines the needed interface to the functions of the extended library.

The class `diffs` is an example of how to store the derivatives using minimal storage. It overloads the `insert` function and stores derivatives which have not, due to symmetry, occured earlier[1]. It then implements the function `get` which can be used to retrieve an arbitrary partial derivative.

[1]It does not use the `calcoffs` function to calculate where to store the derivatives but instead relies on the calling sequence from the extended functions.

B Interval Example

As already mentioned FADBAD can be used to differentiate programs using other types than `double`. One obvious class of types to use with FADBAD is interval types[2]. FADBAD has been successfully tested with the BIAS/PROFIL interval package [Knü93a, Knü93b] and in this section we will see how FADBAD is configured to use the class `INTERVAL` defined in PROFIL as base type in FADBAD.

A file which defines the class `INTERVAL` and the arithmetic operations is shown in File B.1. The file is specified when running the `configure` script and included when the FADBAD library is compiled. The definition of the class `INTERVAL` and some of the basic operations are defined in the header file `Interval.h` while exotic operations are defined in `Functions.h`. Furthermore, because PROFIL uses non-standard function names, we have to redefine the names of the arithmetic functions used in FADBAD to match the names used in PROFIL.

File B.1 The file `interval.h`.

```
#include "Interval.h"
#include "Functions.h"
#define pow Power
#define exp Exp
#define log Log
#define sqrt Sqrt
#define sin Sin
#define cos Cos
#define tan Tan
#define asin ArcSin
#define acos ArcCos
#define atan ArcTan
```

File B.2 is used to tell FADBAD which arithmetic and logical operations are defined on the base type.

The script, `configure` is called to generate a `Makefile` configured for the type of system that the script is executed on. The user is prompted for the base

[2]When using interval types, we are capable of computing mathematical correct enclosures of derivatives.

File B.2 The file `configure.h`.

```
// Configure this file to match the operations that is
// defined in your base type.
// Note that the binary operations + -, and the
// compound assignment operations += -= should always
// be present.

#define HASPOWOPN      // Has pow(BaseType,int)
#define HASMULOP       // Has BaseType*BaseType
#define HASDIVOP       // Has BaseType/BaseType
#define HASPOWOP       // Has pow(BaseType,BaseType)
#define HASUADDOP      // Has +BaseType
#define HASUSUBOP      // Has -BaseType
#define HASEXPOP       // Has exp(BaseType)
#define HASLOGOP       // Has log(BaseType)
#define HASSQRTOP      // Has sqrt(BaseType)
#define HASSINOP       // Has sin(BaseType)
#define HASCOSOP       // Has cos(BaseType)
#define HASTANOP       // Has tan(BaseType)
#define HASASINOP      // Has asin(BaseType)
#define HASACOSOP      // Has acos(BaseType)
#define HASATANOP      // Has atan(BaseType)
#define HASASINOP      // Has asin(BaseType)
#define HASACOSOP      // Has acos(BaseType)
#define HASATANOP      // Has atan(BaseType)
#define HASEQ          // Has BaseType == BaseType
#define HASNEQ         // Has BaseType != BaseType
//#define HASGEQ       // Has BaseType >= BaseType
#define HASLEQ         // Has BaseType <= BaseType
//#define HASGT        // Has BaseType > BaseType
#define HASLT          // Has BaseType < BaseType
#define HASCMULOP      // Has BaseType *= BaseType
#define HASCDIVOP      // Has BaseType /= BaseType
```

type to build the FADBAD library upon, the number of differentiation levels to build, etc. When the `Makefile` has been generated the library can be compiled by typing `make`. See Subsec. 5.1 for details on using the `configure` script.

References

[C++95] ASC, ANSI, AT&T Bell Laboratories, USA. *Working Paper for Draft Proposed International Standard for Information Systems—Programming Language C++*, x3j16/95-0087, wg21/n0687 edition, 1995.

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

[Knü93a] Olaf Knüppel. BIAS – Basic Interval Arithmetic Subroutines. Technical report, Technische Universität Hamburg-Harburg, July 1993.

[Knü93b] Olaf Knüppel. PROFIL – Programmer's Runtime Optimized Fast Interval Library. Technical report, Technische Universität Hamburg-Harburg, July 1993.

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