Automatic differentiation: an implementation in Maple

Dominique Villard and Michael B. Monagan
(e-mail: {villard, monagan}@cecm.sfu.ca)
Dept of Maths & Stats
Simon Fraser University
Burnaby B.C. V5A1S6, Canada

July 22, 1998

Abstract

In this report we present our implementation of automatic differentiation (ie. differentiation of functions represented by programs) in *Maple*: it fully supports the reverse mode and extends differentiation to cope with common linear-algebra operators. It also makes use of standard features of CAS to determine conflicts, optimize computation sequences and memory allocation.

Keywords: automatic differentiation, source transformation, optimization, linear algebra operators, computer algebra system.

Introduction

Automatic Differentiation (AD) has gained an increasing interest over the last decade [13, 9, 4, 8, 16, 15, 1] because computation of derivatives is involved in almost all numerical problems (optimization, adaptative step-size integration, ...) and because divided differences are not accurate nor numerically stable.

Most automatic differentiators are designed to deal with numerical codes written in fortran77 (*Odyssée* [16], *Adifor* [4]) or C (*AdiC*). It may seem weird to differentiate functions written in a language (like *Maple*) whose primary focus is not numerical computations but:

- CAS are widely used to generate automatically systems of equations (describing ODE's, pDE's, ...). In order to solve or integrate them, numerical code is generated and linked to dedicated software (such as *Matlab*).
- they may be used to prototype algorithms. After prototyping numerical "equivalent" code may be generated.

Hence automatic differentiation may take place before the generation of numerical code. Moreover we will see in section 5 that CAS already have the features required for creating a powerful automatic differentiator.

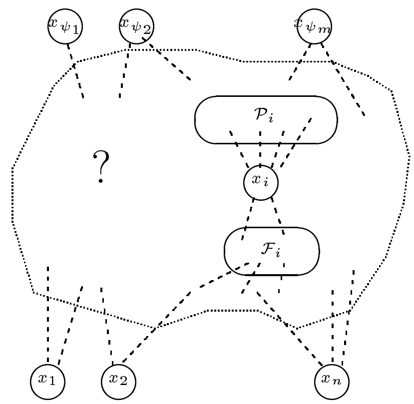
Our paper is organized as follows: in the first section we present the differentiation problem, the modes of differentiation (forward and reverse) and recall their main characteristics. The end of the section is devoted to implementation techniques, ie. how given the code of a function we can compute its derivatives. As it will be shown the code transformation is the most satisfactorily technique. In the second section we present the few rules which are necessary for transforming the code of a function in order to compute its derivatives. The third section is dedicated to some optimizations of the translated code. In the following section we extend the differentiation to cope with some usual linear algebra operators. In the last section a simple example will be presented in order to show how the original function is transformed when using the different modes of differentiation.

1 Differentiation modes

Let us consider a function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ computed by the program given in figure 1-a. Let us also assume that f is a smooth function (ie. that the chain rule applies). An alternative representation of f is the computational graph given in figure 1-b where $\mathcal{F}_i$ is the set of operands involved in the computation of x_i: we have obviously $\mathcal{F}_i \subseteq \{x_1, \ldots, x_{i-1}\}$. $\mathcal{P}_i$ is defined as the set of variables x_j such that $x_i \in \mathcal{F}_j$.

for $i \ := n+1$ **to** V **do**
  $x_i \ := \phi_i(\mathcal{F}_i)$;
done
$f \ := (x_{\psi_1}, \ldots, x_{\psi_m})^T$;

(a) a simple program



(b) its computational graph

Figure 1: two alternate representations of the computation of f

Computing one element of the jacobian matrix (say $\frac{\partial f_r}{\partial x_s}$) leads to evaluate the following expression[1]:

$$\frac{\partial f_r}{\partial x_s} = \frac{\partial x_{\psi_r}}{\partial x_s} = \sum_{\substack{\text{path } \mathcal{P} \\ \text{between } x_{\psi_r} \text{ and } x_s}} \prod_{(x_i,x_j)\in\mathcal{P}} \frac{\partial x_i}{\partial x_j} \tag{1}$$

The modes of differentiation are only ordered ways of computing the transitive-closure like formula (1), but any scheme remains valid (see [10] for an interesting greedy algorithm).

1.1 the Forward mode

This mode [13, 9] is the usual differentiation procedure as it is taught in calculus courses. With this mode let us associate to each variable x_i the linear tangent variables $\hat{x}{}_i^r$ defined by:

$$\hat{x}{}_i^r = \frac{\partial x_i}{\partial x_r} \qquad \forall r \in [1..n] \tag{2}$$

Then the following forward recurrence holds (it is the chain rule):

$$\hat{x}{}_i^r = \frac{\partial \phi_i}{\partial x_r}(\mathcal{F}_i) = \sum_{x_j \in \mathcal{F}_i} \frac{\partial \phi_i}{\partial x_j}.\frac{\partial x_j}{\partial x_r} = \sum_{x_j \in \mathcal{F}_i} \frac{\partial \phi_i}{\partial x_j}.\hat{x}{}_j^r \tag{3}$$

1.2 the Reverse mode

The reverse mode [9, 16] is only a different manner of considering the transitive-like closure formula 1. Let us associate to each variable x_i the adjoint variables $\overset{\vee}{x}{}_i^s$ defined by:

$$\overset{\vee}{x}{}_i^s = \frac{\partial f_s}{\partial x_i} \qquad \forall s \in [1..m] \tag{4}$$

[1] it is not the case when considering non-smooth functions such as abs

Then the $\overset{\vee}{x}{}^s_i$ may be computed using the following backward recurrence:

$$\overset{\vee}{x}{}^s_i = \sum_{x_j \in \mathcal{P}_i} \frac{\partial f_s}{\partial x_j} . \frac{\partial x_j}{\partial x_i} = \sum_{x_j \in \mathcal{P}_i} \overset{\vee}{x}{}^s_j . \frac{\partial x_j}{\partial x_i} = \sum_{x_j \in \mathcal{P}_i} \overset{\vee}{x}{}^s_j . \frac{\partial \phi_j}{\partial x_i} \tag{5}$$

In practice the formula (5) is not very useful since it is much harder for a variable to know the operations where it is involved (the set $\mathcal{P}_i$) than "its" operands (the set $\mathcal{F}_i$). So we will rather use the following alternative formula:

$$\overset{\vee}{x}_j \mathrel{+}= \frac{\partial \phi_i}{\partial x_j} \overset{\vee}{x}_i \quad \forall x_j \in \mathcal{F}_i \tag{6}$$

which is equivalent to (5) provided that adjoint variables are initialized to 0.

1.3 A brief comparison

The table 1 compares the relative properties of the forward and the reverse mode. In that table $T(\ldots)$ (resp. $S(\ldots)$) represents the "time" (resp. the "space") required for computing .... f (resp. $\mathbf{f}$) is assumed to be a function of $\mathbb{R}^n \mapsto \mathbb{R}$ (resp. $\mathbb{R}^n \mapsto \mathbb{R}^m$)

From an analysis of this table we can make the following remarks:

- the time required to compute the derivative of a function varies proportionally wrt. the size of its inputs when applying the forward mode. It varies proportionally wrt. the size of its outputs when the reverse mode is used. Hence we have the very nice property that (theoretically) the gradient of a function may be be computed in a time proportional to the time of computing the function itself. This result is one of the principal motivations of research in AD and in the increasing use of the reverse mode in optimization problems. The time complexity for computing jacobians may be lowered when sparse representations are used or when the sparsity pattern of the jacobian is known: in such case a "compressed" jacobian may be computed more efficiently [3].
- the space required by the forward mode varies with the size of the inputs of the function. This complexity is also lowered when compressed jacobians are computed. The space complexity varies mainly with the number of operations performed (usually $T(\mathbf{f}) \gg S(\mathbf{f})$) when using the reverse mode: this is the major drawback of this mode. A. Griewank [11] proposed a strategy to reduce this complexity but at the expense of degradation in the time complexity (due to partial recomputations).

	Finite Differences	Forward Mode	Reverse Mode
$T(f, \nabla f)$	$(n+1).T(f)$	$\leq 4n.T(f)$	$\leq 4.T(f)$
$T(f, \nabla f, H.y)$		$\mathcal{O}(n.T(f))$	$\leq 14.T(f)$
$S(f, \nabla f)$	$S(f)$	$\mathcal{O}(n.S(f))$	$\mathcal{O}(S(f) + T(f))$
$T(\mathbf{f}, J.y)$		$\mathcal{O}(T(\mathbf{f}))$	$\leq 4m.T(\mathbf{f})$
$T(\mathbf{f}, z^T.J)$		$\mathcal{O}(n.T(\mathbf{f}))$	$\leq 4.T(\mathbf{f})$
$T(\mathbf{f}, J)$	$(n+1).T(\mathbf{f})$	$\mathcal{O}(n.T(\mathbf{f}))$	$\leq (3m+1).T(\mathbf{f})$
$S(\mathbf{f}, J)$	$S(\mathbf{f})$	$\mathcal{O}(n.S(\mathbf{f}))$	$\mathcal{O}(m.S(\mathbf{f}) + T(\mathbf{f}))$

Table 1: Comparison of differentiation modes

1.4 Implementation Techniques

In this subsection we discuss the different ways of obtaining the derivatives. The first technique is to compute the graph of sensitivities at runtime. This technique has two major drawbacks: the graph may be huge and its evaluation is slow (interpreted computation). Moreover if higher derivatives must be provided, local sensitivities are not sufficient: one must also keep track of the operations performed. This model is implemented in *Jakef* and *AdolC/AdolC++*.

The second technique is to use operator and function overloading (when the language permits it) to perform the computation of the function and of its derivatives at the same time. This method, used in Pascal-SC [14] is the easiest to implement (only variable types are changed). It has two major drawbacks: it is restricted to the forward mode and derivatives can only be provided up to a certain order (depending on the implementation).

The last technique is the source to source translation. The source code is augmented (by adding some instructions) in order to compute the initial function and its derivatives. This technique, implemented in *Odyssée* [16], *Adifor* [4] and *Maple* [15] is the most elegant technique. Moreover under some assumptions on the kind of transformation applied the code produced may be reintroduced in the differentiator in order to obtain higher derivatives (but this technique is less efficient than producing all derivatives in one pass [1]: for example one can not take advantage of the symmetries).

2 Code Transformation

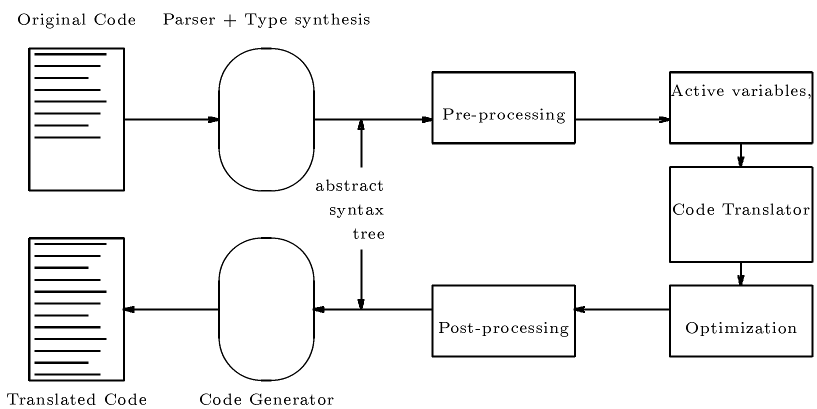


Figure 2: Code Transformation processus

The figure 2 depicts the different steps which are performed during the code translation. They consist in:

1. parsing: the text of the input function is transformed to a structure (abstract syntax tree) which is suitable for manipulation. This step is performed using the `maple2intrep` command.
2. type analysis: at this stage we analyse the algebraic expressions, function calls and indexed variables to determine the type of variables. For automatic differentiation the main goal of this stage is to find which variables are scalars and which are vector in order to compute the size of the derivatives and to initialize adjoint variables properly. Moreover that knowledge is crucial to select the "most" efficient differentiation mode.
3. pre-processing: for example a correct Maple expression is `evalm( A * x)` which performs scalar-matrix multiplication. To perform correctly automatic differentiation we need to know which is the scalar between `A` and `x` (in that case the code would be transformed to `scalarmul(A, x)`. When using the reverse mode some statements will have to be transformed so that the code meets the requirements for proper differentiation (see subsection 2.5.2).
4. dependencies computation: at this stage we compute the dependencies between variables in order to compute which ones are "active" (ie. participate to the computation of the function's value).
5. code transformation: this stage is the core of the processus. Each statement and statement sequences are analysed and transformed according to user requirements: these transformations are described in the following subsection. During this stage some optimizations of the code may be performed.
6. optimization: this stage is not implemented yet. Its purpose will be to rearrange code and to remove code which is unnecessary ("dead" code or unncessary resets of variables).

7. post-processing: when we have differentiated the function we may get a code which is syntactically correct but which could not be executed (see for example the differentiation of the cross-product in section 4): this is because differentiation of an expression is performed (mostly) syntactically without information on the type of variables handled.
8. code generation: at this stage we translate the modified abstract syntax tree back to a Maple function. This is performed using the pre-defined command `intrep2maple`.
   Remark: there is no reason for the language of the source-code to be the same than the language of the ouput-code. We could also generate C or Fortran77 code.

2.1 Transformation rules

In this section we present the simple transformations applied to the basic constructions of all imperative languages. As the reader will remark in the remainder of this section the code transformation technique is far more complicated to implement with the reverse mode than with the forward mode. These rules are the most general ones and may be specialized according to the context and/or user-requirements in order to generate a more efficient code (see exemple in section 5.1).

For the forward mode we can proceed in one pass: as shown by the recurrence relation (3), the variable x_i and its linear tangent variables $\hat{x}{}_i^r$ may be computed at the same time (they depend on variables already computed). We will denote TLC(S) the (linear tangent) code obtained after transformation of a statement S.

For the reverse mode things are quite different: the recurrence relations (5) and (6) show that the adjoint variable $\overset{\vee}{x}{}_i^s$ may be computed only once the adjoint variables $\overset{\vee}{x}{}_j^s$ are known. Hence we must proceed in two stages: during the first stage the computation of the function is realized and partial derivatives are stored (or the variables involved in their evaluations). The second stage performs the computation of adjoints variables. We will see in the sequel that the reverse mode places much more constraints on the source code than the forward mode. In the remainder of this section we will note T(S) the augmented code that computes the original function and A(S) the code that performs propagation of adjoint variables.

Note: In the following the Push(...) and Pop() commands are used to specify that a value must be stored on (resp. retrieved from) a stack (in fact only non constant values will be stored).

2.2 Assignment statement

Let us assume that the statement has the form:

$$y := f(x_1, \dots, x_k, \{y,\}c_1, \dots, c_l)$$

where f is a scalar algebraic expression involving only elementary operators (+,-,*,/,$\hat{}$) and functions (sin, cos, ...), y and the x_i's are scalar active variables, and the c_i are constants or inactive variables.

Note: the following transformations may be easily extended to cope with multi-assignment (as in *Maple* or *Matlab* for exemple).

2.2.1 Linear tangent code

Here we present the algorithm refered as DAD1 in [8]. This is the direct application of the recurrence formula (3):

$$\begin{array}{l} \nabla y := \sum_{x_i} (\nabla x_i * df/dx_i) \ \{+ \ \nabla y * df/dy \ \} \\ y := f(x_1, \dots, x_k, \{y,\}c_1, \dots, c_l) \end{array}$$

where $\nabla x_i = \left[\hat{x}{}_i^1, \dots, \hat{x}{}_i^n\right]$. In a sequential implementation the computation of ∇y must precede the computation of y since y may be on both sides of the assignment. Alternative algorithms (refered as DAD2 and DAD3 in [8]) exist but both lead to increase the size of the code, provide no complexity gain and are more difficult to implement. Moreover DAD2 can not take advantage of a possible sparsity.

2.2.2 Adjoint code

Here we present the algorithm refered as DAI1 in [8]. This is the application of the recurrence formula (6):

T(S)	A(S)
{Push(dy/dy);}	$\check{X}_k$ += Pop() * $\check{Y}$;
Push(dy/dx_1);	$\vdots$
$\vdots$	$\check{X}_1$ += Pop() * $\check{Y}$;
Push(dy/dx_k);	{$\check{Y}$:= Pop() * $\check{Y}$;} or $\check{Y}$:= 0_m;
$y := f(x_1, \ldots, x_k, \{y, \} c_1, \ldots, c_l)$	

where $\check{X}_i = \left[\check{x}_i^1, \ldots, \check{x}_i^m\right]$.

Warning: with the reverse mode one must be careful if both left and right hand sides use the same indexed variable. In such case one must make sure that the indices are distinct. This problem is closely related to the problem of computing true dependencies in the parallelisation of a code [2]. Here again a computer algebra system is very useful to check wether there may be a conflict.

2.3 Sequence of statements

Let us consider the following statement sequence S:

$$S_1;\ S_2; \ldots;\ S_{n-1};\ S_n$$

2.3.1 Linear tangent code

For the forward mode this piece of code will give rise to:

$$\mathrm{LTC}(S_1);\ \mathrm{LTC}(S_2);\ \ldots;\ \mathrm{LTC}(S_{n-1});\ \mathrm{LTC}(S_n)$$

2.3.2 Adjoint code

For the reverse mode S will be transformed into the two following statement sequences (remark the reverse order for A(S)):

T(S)	A(S)
$\mathrm{T}(S_1);\ \mathrm{T}(S_2);\ \ldots;\ \mathrm{T}(S_{n-1});\ \mathrm{T}(S_n)$	$\mathrm{A}(S_n); A(S_{n-1}); \ldots; A(S_2); A(S_1)$

2.4 If Statement

Let us consider the following conditional statement S:

if C_1 **then** S_1
elif C_2 **then** S_2
$\vdots$
elif C_{n-1} **then** S_{n-1}
{**else** S_n }
fi

We will assume that none of the conditions C_i has side effects.

2.4.1 Linear tangent code

For the forward mode we simply transform each branch of the statement S. Hence LTC(S) is:

```
if C₁ then LTC(S₁)
elif C₂ then LTC(S₂)
⋮
elif Cₙ₋₁ then LTC(Sₙ₋₁)
{else LTC(Sₙ) }
fi
```

2.4.2 Adjoint code

With this mode we need to remember which branch was executed in order to properly propagate derivatives:

T(S)	A(S)
b := 0;	b := Pop();
if C_1 **then** b := 1; T(S_1)	**if** b = 1 **then** A(S_1)
elif C_2 **then** b := 2; T(S_2)	**elif** b = 2 **then** A(S_2)
⋮	⋮
elif C_{n-1} **then** b := n-1; T(S_{n-1})	**elif** b = n-1 **then** A(S_{n-1})
{**else** b := n; T(S_n) }	{**else** A(S_n) }
fi ;	**fi**
(∗) Push(b);	

Note: this transformation is correct iff all after all T(S_i) the statement labelled (∗) is executed. If some statement T(S_i) (for example if the behavior of a loop is controlled by **break** statements) do not branch to this one some code must be added to T(S_i) to ensure that b is saved.

2.5 Loop Statement

Let us consider the following general loop:

for i **from** m **to** M **by** s **while** c **do**
    statement sequence
od

We will assume that the computations of starting index, ending index and step do not have side effects.

2.5.1 Linear tangent code

With the forward mode we just need to augment the body of the loop:

for i **from** m **to** M **by** s **while** c **do**
    LTC(statement sequence)
od

2.5.2 Adjoint code

For the reverse mode we will also assume that starting index m and step s are constant over the loop.

T(S)	A(S)
for i **from** m **to** M **by** s **while** c **do**	i, m, s := Pop();
T(statement sequence)	**for** i **from** i-s **to** m **by** -s **do**
od	A(statement sequence)
Push(i, m, s);	**od**

This transformation is valid if the loop S ends normally (ie. if its behaviour is not controlled by break statements). One can easily extend that transformation to cope with break statements occurring at the beginning or at the end of a "block". If we had allowed break statements to occur anywhere in the body of a loop it would have forced us to use goto statements in the propagation code A(S) and the final code would be messy and dirty (jumps in the middle of a loop!). Hence in our implementation it is up to the preprocessing stage to transform the original code so that it conforms to the requirements.

2.6 Some remarks

As the reader may realize the forward mode is quite straightforward to implement while the forward mode is much more technical to implement. Moreover the reverse mode often requires extra space for computing derivative, and this space may be dynamic space which is less efficient.

3 Optimizations

One can distinguish three main goals for optimizations:

- decrease the memory requirements especially for the reverse mode;
- decrease the overall complexity of differentiation;
- reduce (possibly void) the requirement of dynamic memory: the purpose here is to allow the optimizer of a compiler to perform some tasks such as loop/rearrangment, vectorization, parallelization.

In this paper the last two points are discussed.

3.1 Computation sequence

Expression splitting [6, 15] had been used to generate more efficient codes. If we apply such transformation of the code it may degradate the time complexity (because the total in-degree increases) and it may require more additional memory when using the reverse mode. We also can no longer apply the greedy algorithm [10] to obtain more efficient code. Moreover, its main disavantage lies in the fact that we may lose significant information on the differentiability of the function. For example let us consider the function $f : x \mapsto |x^3|$. f is differentiable everywhere. One may split the evaluation of f as `u := x**3; f := abs(u)`. Now if we compute the derivative of f wrt. u this derivative has a singular point at $u = 0$. We have lost the differentiability of f.

When we compute the derivatives of a simple assignment statement (ie. we compute a gradient in the scalar case) the expression that represent the right-hand side is a formula. Obviously this latter may be represented by a DAG, which may be differentiated with respect to the active variables. The result of this step is a DAG and both DAGs may be used to generate an optimized computation sequence [2, 15].

3.2 Memory

We have seen in the presentation of the transformation rules (see section 2.1) that the reverse mode requires additional storage (in fact a stack) to save some variables or some control information. Apart from time consuming, dynamic storage has a major drawback: code-optimization techniques [2] can no longer be used to transform the generated code since different transformations applied to T(S) and A(S) would lead to erroneous computations. Hence when the size of the extra-storage may be statically computed we rather use static storage (arrays). For example let us consider the following loops obtained after transformation of a loop S:

T(S)	A(S)
for i **from** m **to** M **by** s **while** c **do**	**for** i **from** i-s **to** m **by** -s **do**
⋮	⋮
Push(x);	e := Pop();
⋮	...e...
od	⋮
	od

If we know how many times the statement Push(x); is executed (or an upper bound) we can allocate static memory to store the successive values of x. For example if we assume that Push(x); is not in a inner loop then we know that it will be executed at most $\lfloor \frac{M-m}{s} \rfloor + 1$ times (note: we can perform the same kind of computations for inner-loops). Hence we can transform the code as follows:

T(S)

```
e := vector( iquo(M-m, s)+1);
for i from m to M by s while c do
    ⋮
    e[1 + iquo(i-m, s)] := x;
    ⋮
od
```

A(S)

```
for i from i-s to m by -s do
    ⋮
    ... e[1 + iquo(i-m, s)] ...
    ⋮
od
```

which uses only static storage.

4 Dealing with vector code

If some years ago programs were built from scratch, nowadays people use libraries (such as BLAS, Lapack, IMSL, *Matlab*,...) to build quickly complex programs. Apart from rapidity processing in such way offers many advantages:

- very often these libraries are robust.
- somes functions may be coded differently regarding the target architecture (sequential, parallel or vectorized);

The table 2 reports the extension of differentiation to linear algebra operators. In this table A, B and C are assumed to be matrices (or vectors), u, w and w are assumed to be vectors and x is assumed to be a scalar. The $.*$ operator and cumsum function are borrowed from *Matlab* (array multiplication and cumulative sum). Here we also assume that we differentiate wrt. a single scalar (forward mode) or to a scalar function (reverse mode).

operation	notation	forward mode	reverse mode
addition	$C = A + B$	$\hat{C} = \hat{A} + \hat{B}$	$\check{A} \mathrel{+}= \check{C},\ \check{B} \mathrel{+}= \check{C}$
multiplication	$C = AB$	$\hat{C} = \hat{A}\, B + A\, \hat{B}$	$\check{A} \mathrel{+}= \check{C}\, {}^tB,\ \check{B} \mathrel{+}= {}^tA\, \check{C}$
scalar-matrix multiplication	$C = xA$	$\hat{C} = x\, \hat{A} + \hat{x}\, A$	$\check{A} \mathrel{+}= x\, \check{C},\ \check{x} \mathrel{+}= \mathrm{sum}(A.*\, \check{C})$
(hermite-)transposition	$C = {}^tA$	$\hat{C} = {}^t\hat{A}$	$\check{A} \mathrel{+}= {}^t\check{C}$
cross product	$w = u \wedge v$	$\hat{w} = \hat{u} \wedge v + u \wedge \hat{v}$	$\check{u} \mathrel{+}= v \wedge \check{w},\ \check{v} \mathrel{+}= \check{w} \wedge u$
dot product	$x = u.v$	$\hat{x} = \hat{u}.v + u.\hat{v}$	$\check{u} \mathrel{+}= \check{x}\, v,\ \check{v} \mathrel{+}= \check{x}\, u$

Table 2: differentiation of vector operators

The main difficulty when dealing with vector codes is that multiplication is no longer commutative: hence conversely to the scalar case the computation of local derivatives can no longer be separated from the computation of tangent or adjoint variables.

5 An Implementation in Maple

As shown by the figure 3 *Maple* contains most of the features that we expect from an automatic differentiator:

- as a computer algebra system, the ability to perform differentiation of formulae is a basic feature.
- some tools exist for converting a subset of Maple functions to C or Fortran77 numerical code. As a matter of fact these tools rely upon other tools that convert a Maple function to an abstract syntax tree.
- piecewise and piecewise-like (such as abs, max,...) expressions may be differentiated. In that case *Maple* uses the limit definition to find out wether the expression is differentiable or not. Hence one of the most important problems that may occur using automatic differentiation [7] is partially solved.

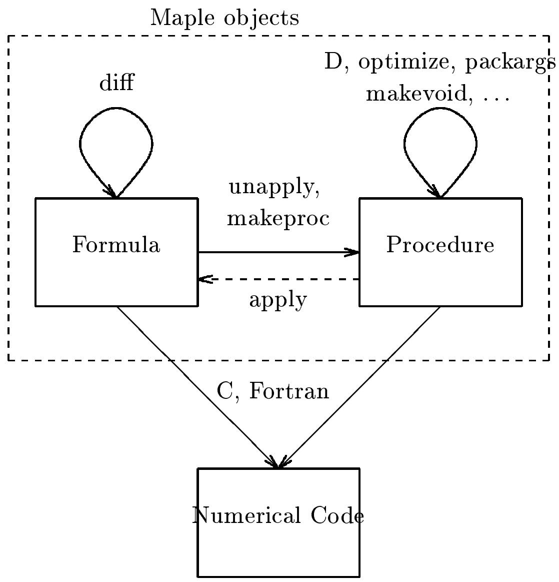


Figure 3: converting tools in Maple

The missing links on the figure are that we are unable to generate Maple code from other languages. The table 3 presents the current abilities of *Maple* with respect to automatic differentiation. The bullets represent the abilities of the commercial version while the circles represent the specific new features. By Hybrid differentiation we mean a model where the software selects the "most" efficient mode depending on the context (*Adifor* may be viewed as an hybrid differentiator since it uses the reverse mode to differentiate expressions and the forward mode to propagate derivatives).

	Forward mode	Reverse mode	Hybrid
Formula	$\bullet$	$\circ$	$\circ$
Straight Line code	$\bullet$	$\bullet$	$\circ$
Conditional	$\bullet$	$\bullet$	$\circ$
Loops	$\bullet$	$\circ$	$\circ$
Function call	$\bullet$	$\circ$	$\circ$
Recursive proc	$\bullet$	$\bullet$	
Higher order	$\bullet$	$\bullet$	
Non-scalar inputs	$\circ$	$\circ$	
Vector code	$\circ$	$\circ$	
Error estimation			

Table 3: Maple Capabilities wrt. Automatic Differentiation

5.1 A short example: Legendre polynomials

Let us consider the derivatives of Legendre polynomials [5] which are defined recusively as follows:

$$P_0(x) = 1$$

$$
\begin{aligned}
P_1(x) &= x \\
(n+1)P_{n+1}(x) &= (2n+1)\,x\,P_n(x) - n\,P_{n-1}(x) \qquad \forall n \geq 2
\end{aligned}
$$

These polynomials may be computed using the following function:

```
> print( Legendre);
```

```
proc(m::integer, x::algebraic)
    local n, pn, pnm1, pnm2;
      if m = 0 then pn := 1
      elif m = 1 then pn := x
      else
          pnm2 := 1;
          pnm1 := x;
          for n from 2 to m do
              pn := ((2 × n − 1) × x × pnm1 − (n − 1) × pnm2)/n;
              pnm2 := pnm1;
              pnm1 := pn
          od
      fi;
      RETURN(pn)
    end
```

First we use the forward mode to generate a function that computes the derivative of the Legendre polynomials:

```
> forwardLegendrePrime := Differentiate( Legendre, [x], "Mode" = "forward");
```

```
forwardLegendrePrime := proc(m::integer, x::algebraic)
    local Legendrelt, n, pn, pnlt, pnm1, pnm1lt, pnm2, pnm2lt;
    description`Generated by Automatic differentiation of : Legendre  Wed Ju\
    n 10 16 : 25 : 16 PDT 1998\n`;
      if m = 0 then pnlt := 0; pn := 1
      elif m = 1 then pnlt := 1; pn := x
      else
          pnm2lt := 0;
          pnm2 := 1;
          pnm1lt := 1;
          pnm1 := x;
          for n from 2 to m do
              pnlt := pnm1lt × (2 × n − 1) × x/n + pnm2lt × (−n + 1)/n + ((2 × n − 1) × pnm1)/n;
              pn := ((2 × n − 1) × x × pnm1 − (n − 1) × pnm2)/n;
              pnm2lt := pnm1lt;
              pnm2 := pnm1;
              pnm1lt := pnlt;
              pnm1 := pn
          od
      fi;
      Legendrelt := pnlt;
      RETURN(Legendrelt)
    end
```

Now we generate a second function using the reverse mode:

```
> reverseLegendrePrime := Differentiate( Legendre, [x], "Mode" = "reverse");
```

```
reverseLegendrePrime := proc(m::integer, x::algebraic)
    local n, pn, pnad, pnm1, pnm1ad, pnm2, pnm2ad, res, resad, ts, xad;
    description`Generated by Automatic differentiation of : Legendre  Wed Ju\
    n 10 16 : 25 : 47 PDT 1998\n`;
    # initializations
```

```
(*)       ts := array(1.. - 1 + m) ;
          pnad := 0 ;
          pnm1ad := 0 ;
          pnm2ad := 0 ;
          xad := 0 ;
          resad := 1 ;
      # progression stage
          if m = 0 then pn := 1
          elif m = 1 then pn := x
          else
              pnm2 := 1 ;
              pnm1 := x ;
              for n from 2 to m do
                  ts[n-1] := ((2 × n - 1) × pnm1)/n ;
                  pn := ((2 × n - 1) × x × pnm1 - (n - 1) × pnm2)/n ;
                  pnm2 := pnm1 ;
                  pnm1 := pn
              od
          fi;
          res := pn ;
      # reverse propagation stage
          pnad := pnad + resad ;
          resad := 0 ;
          if m = 0 then pnad := 0
          elif m = 1 then xad := xad + pnad ; pnad := 0
          else
              for n from m by - 1 to 2 do
                  pnad := pnad + pnm1ad ;
                  pnm1ad := 0 ;
                  pnm1ad := pnm1ad + pnm2ad ;
                  pnm2ad := 0 ;
                  xad := xad + ts[n-1] × pnad ;
                  pnm2ad := pnm2ad - ((n - 1) × pnad)/n ;
                  pnm1ad := pnm1ad + ((2 × n - 1) × x × pnad)/n ;
                  pnad := 0
              od;
              xad := xad + pnm1ad ;
(-)           pnm1ad := 0 ;
(-)           pnm2ad := 0
          fi;
          RETURN(xad)
      end
```

We can see on this example that the code generated with the reverse mode is longer than the code generated with the forward mode. This is because:

- each construction of the language gives rise to two constructions in the derived code (one for the progressive stage, one for the reverse stage).
- adjoint variables must be initialized and reset. Some of these resets may give rise to "dead-code" [2] (see for example the lines labelled $(-)$). A future version of the software will remove some of these initializations and resettings.

For the code generated using the reverse mode we can also remark that the software was able to compute the size of additional memory required: hence rather than dynamic memory, static memory is used (see line labelled $(*)$). Moreover no variable is required to store the branch used in conditional statements since the conditions ($m = 0$, $m = 1$) are constant.

Once we have generated these functions we can check that they compute properly the derivatives of the Legendre polynomials by comparing their evaluations at a formal point `x` to the derivative of the legendre polynomial at `x` obtained by symbolic differentiation:

```
> p4 := Legendre(4, x);
```

$$p4 := \frac{7}{4}\,x\,(\frac{5}{3}\,x\,(\frac{3}{2}\,x^2 - \frac{1}{2}) - \frac{2}{3}\,x) - \frac{9}{8}\,x^2 + \frac{3}{8}$$

```
> pp4s := diff( p4, x);
```

$$pp4s := \frac{35}{12}\,x\,(\frac{3}{2}\,x^2 - \frac{1}{2}) - \frac{41}{12}\,x + \frac{7}{4}\,x\,(\frac{15}{2}\,x^2 - \frac{3}{2})$$

```
> pp4f := forwardLegendrePrime( 4, x);
```

$$pp4f := \frac{35}{12}\,x\,(\frac{3}{2}\,x^2 - \frac{1}{2}) - \frac{41}{12}\,x + \frac{7}{4}\,x\,(\frac{15}{2}\,x^2 - \frac{3}{2})$$

```
> pp4r := reverseLegendrePrime( 4, x);
```

$$pp4r := \frac{35}{12}\,x\,(\frac{3}{2}\,x^2 - \frac{1}{2}) - \frac{7}{3}\,x + \frac{7}{4}\,(\frac{5}{2}\,x^2 - \frac{5}{6})\,x + 3\,x\,(-\frac{3}{4} + \frac{35}{12}\,x^2)$$

We can remark that the polynomial `pp4f` is strictly the same as the polynomial `pp4s` (obtained by symbolic differentiation of `p4`): it comes with no surprise since both are obtained using the forward mode. Using the reverse mode we obtain a polynomial which is different from the previous ones, but we can easily check that in fact `pp4r` is equivalent to `pp4s`:

```
> simplify( pp4r - pp4s);
```

$$0$$

The main interest of automatic differentiation is illustrated afterwards. We can now compute the first derivative of any Legendre polynomial at any point very efficiently. For example we can compute $P'_{500}(0.9)$:

```
> reverseLegendrePrime( 500, 0.9);
```

$$-58.65581865$$

Note: we could reintroduce the output code in the software in order to obtain a code that would compute second derivatives and do it again and again but the size of the code would grow exponentially with the order of differentiation.

More examples can be accessed on the Web:

http://www.cecm.sfu.ca/~villard/AD/determ/determ1.html : shows the derivation of a function that computes the determinant of a matrix in order to compute its "gradient", and

http://www.cecm.sfu.ca/~villard/AD/groten/groten1.html : in that example we compute the gradient of a function where vector operations are involved (scalar-matrix multiplications, dot and cross products). This function comes from a bio-chemistry optimization problem.

6 Conclusion

In this paper we presented the automatic differentiation problem from a code transformation viewpoint. We shown the basic transformation rules that are applied to the body of a procedure in order to compute its derivatives. We also extended the differentiation functions in order to deal with procedures involving linear-algebra operations. Finally we presented a short application.

References

[1] J. Abate, C. Bischof, and L. Roh, *Algorithms and Design for a second-order Automatic Differentiation Module*, in ISSAC'97, W. W. Küchlin, ed., 1997, pp. 149–155.

[2] A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools*, Addison-Wesley Publishing company, 1986.

[3] B. Averick, J. Moré, C. Bischof, A. Carle, and A. Griewank, *Computing large sparse jacobian matrices using automatic differentiation*, SIAM Journal on Scientific Computing, 15(2):285–294, 1994.

[4] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, *Adifor - generating derivatives codes form fortran programs*, Tech. Rep. MCS-P263-0991, Argonne National Laboratory, 1991.

[5] W.H. Beyer, *Standard mathematical tables*, CRC Press, 1978.

[6] C. Faure, *Splitting of Algebraic Expressions for Automatic Differentiation*, in computational differentiation: techniques, applications, and tools, M. Berz, C. Bischof, G. Corliss and A. Griewank, eds., 1996, pp. 117–128.

[7] H. Fisher, *Special problems in automatic differentiation*, in automatic differentiation of algorithms : theory, implementation, and application, A. Griewank and G. Corliss, eds., 1991, pp. 43–50.

[8] J. C. Gilbert, G. Le Vey, and J. Masse, *La différentiation automatique de fonctions représentées par des programmes*, Tech. Rep. 1557, INRIA, Nov. 1991.

[9] A. Griewank, *On automatic differentiation*, Tech. Rep. ANL/MCS-P10-1088, Mathematics and Computer Science Division of Argonne National Laboratory, Nov. 1988.

[10] A. Griewank, and S. Reese, *On the calculation od jacobian matrices by the Markowitz rule*, in automatic differentiation of algorithms : theory, implementation, and application, A. Griewank and G. Corliss, eds., 1991, pp. 126–135.

[11] A. Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Tech. Rep. ANL/MCS-P228-0491, Mathematics and Computer Science Division of Argonne National Laboratory, Apr. 1991.

[12] M. Iri, *History of automatic differentiation and rounding error estimation*, in automatic differentiation of algorithms : theory, implementation, and application, A. Griewank and G. Corliss, eds., 1991, pp. 3–16.

[13] L. B. Rall, *Automatic differentiation - techniques and applications*, in Lecture Notes in Computer Science, no. 120 in VIII, Springer-Verlag, 1981.

[14] ———, *Differentiation in pascal-sc : Type gradient*, ACM Transactions on Mathematical Software, 10 (1984), pp. 161 – 184.

[15] M. B. Monagan, and René. R. Rodoni, *An implementation of the Forward and Reverse Modes of Automatic Differentiation in Maple*, in computational differentiation: techniques, applications, and tools, M. Berz, C. Bischof, G. Corliss and A. Griewank, eds., 1996, pp. 353–362.

[16] N. Rostaing-Schmidt, *Différentiation automatique : application à un problème d'optimisation en météorologie*, PhD thesis, Université de Nice-Sophia Antipolis, Dec. 1993.

[17] D. Villard, *D'un formalisme énergétique à la résolution des équations du mouvement : l'usage de la différentiation et des graphes*, PhD thesis, Université de Rennes 1, Dec. 1995.