GRADIENT: Algorithmic Differentiation in Maple

Michael B. Monagan
Institut für Wissenschaftliches Rechnen,
ETH Zürich, CH–8092 Zürich, Switzerland.
monagan@inf.ethz.ch

Walter M. Neuenschwander
Institut für Kommunikationstechnik,
ETH Zürich, CH–8092 Zürich, Switzerland.
waneu@vision.ethz.ch

Abstract

Many scientific applications require computation of the derivatives of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ as well as the function values of f itself. All computer algebra systems can differentiate functions represented by formulae. But not all functions can be described by formulae. And formulae are not always the most effective means for representing functions and derivatives.

In this paper we describe the algorithms used by the Maple [2] routine `GRADIENT` that accepts as input a Maple procedure for the computation of f and outputs a new Maple procedure that computes the gradient of f. The design of the `GRADIENT` routine is such that it is also trivial to generate Maple procedures for the computation Jacobians and Hessians.

Keywords: Algorithmic Differentiation, Chain Rule, Computer Algebra, Gradients, Jacobians, Hessians.

1 Introduction

What do we understand by *Algorithmic Differentiation* ? What we want to do is to compute the partial derivatives of a function f with respect to an arbitrary set of variables where the function is defined by a program instead of a formula. The program may be a general algorithm containing conditional branches and loops.

The basic idea behind the *Algorithmic Differentiation* of programs is very simple. Applying the elementary rules of differentiation to each statement in the computation-sequence of f yields a new sequence for the computation of the desired derivatives. The key to this method is the repeated use of the chain rule [10].

The very first approaches in *Algorithmic Differentiation* were done more than 30 years ago [1, 17]. Since then numerous implementations of *Algorithmic Differentiation* have have been done primarily to differentiate Fortran and C functions. The paper of D. Juedes [8] gives a summary of all tools developed up till now.

The techniques of *Algorithmic Differentiation* (also called *Automatic Differentiation*) are increasingly used in numerical computation and applied-analytical methods. One of the advantages of *Algorithmic Differentiation* is accuracy. The traditional numerical methods based on *Divided Differences* are often inaccurate. Derivatives of order 3 and higher can be completely useless. Another is generality, and a third advantage is efficieny. Here is a summary of the advantages

- the numerical evaluation of the generated computation-sequences is much more accurate and faster than the methods based on *Divided Differences* (see [3] for a comparison of these methods)
- the representation of the function and its derivatives as a computation-sequence is in general far more compact than a formula or even a DAG (Directed Acyclic Graph).
- *Algorithmic Differentiation* is more general in that we can potentially handle a function defined by an arbitrary program
- P. Wolfe [18] showed that the cost of evaluating a gradient with n components is uniformly bounded and doesn't depend on the number n of independent variables.
  Hence *Algorithmic Differentiation* is really a powerful tool for generating effective code for gradient–calculations

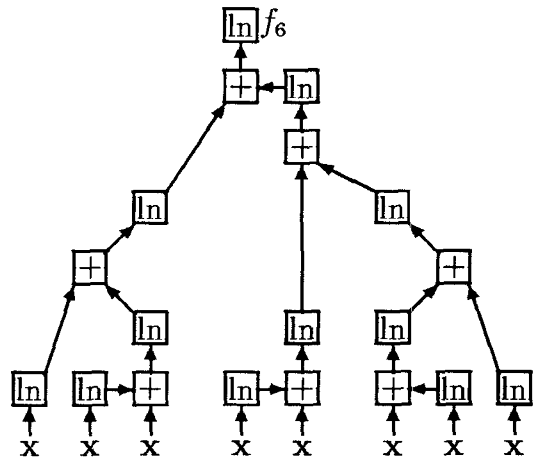
We want to illustrate some of these advantages by looking at an example. Consider the function f_n defined by $f_0 = 0, f_1 = x$ and $f_n = \ln(f_{n-1} + f_{n-2})$ for $n \geq 2$.

In the following Maple code, we have defined f and computed $f_6(x)$ and $f_6'(x)$.

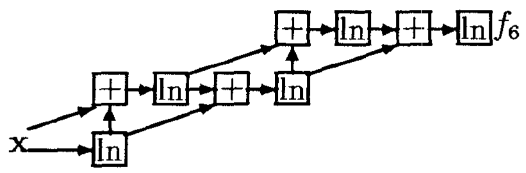
```
> f := n -> ln(f(n - 1) + f(n - 2)):
> f(0) := 0: f(1) := x:
> lprint(f(6));
ln(ln(ln(ln(ln(x)+x)+ln(x))+ln(ln(x)+x))+
ln(ln(ln(x)+x)+ln(x)))

> lprint(diff(f(6),x));
((((1/x+1)/(ln(x)+x)+1/x)/(ln(ln(x)+x)+ln(x))+
(1/x+1)/(ln(x)+x))/(ln(ln(ln(x)+x)+ln(x))+ln(ln(
x)+x))+((1/x+1)/(ln(x)+x)+1/x)/(ln(ln(x)+x)+ln(x
)))/(ln(ln(ln(ln(x)+x)+ln(x))+ln(ln(x)+x))+ln(ln
(ln(x)+x)+ln(x)))
```

The output from Maple shows the representation of $f_6(x)$ and its derivative by a formula printed in one dimension. Systems that operate on these formulas are very ineffective because the size of the formula for $f_n(x)$ and its derivatives grows very large as a function of n due to repeated common subexpressions that are produced under differentiation. We need a better representation for $f_6(x)$.
Another way to represent $f_6(x)$ is by a DAG (Directed Acyclic Graph) which looks like this:



If repeated sub-expressions are shared, the DAG looks like this



The Maple system always optimizes the storage of DAG's in this way. It's obvious that this representation is far more compact. Hence a differentiation based on this shared DAG will be faster. The running time of the Maple `diff` function is proportional to the size of the optimized DAG. In general the gain made by using a DAG instead of a formula is an exponential factor. But an even more compact representation for f, independent of n, is to represent f as a program. For example, the corresponding Maple–program for f is

```
> f := proc(x,n) local a,b,h;
>    if n=0 then 0
>    else
>       a := 0;
>       b := x;
>       to n-1 do h := b; b := ln(a+b); a := h od;
>       b
>    fi
> end:
```

Computing the derivative of f as a program we obtain

```
> GRADIENT(f,[x]);

proc(x,n)
local a,b,da,db,dh,h;
   if n = 0 then 0
   else
      da := 0;
      a := 0;
      db := 1;
      b := x;
      to n-1 do
         dh := db;
         h := b;
         db := (da+db)/(a+b);
         b := ln(a+b);
         da := dh;
         a := h
      od;
      db
   fi
end
```

The resulting Maple procedure is quite compact even for arbitrary n! The cost of evaluating the function f' and the complexity of the corresponding non-optimized DAG will increase by an exponential factor if n increases. Whereas the complexity and the length of the program–representation won't change!

The advantage of using a computer algebra system for *Algorithmic Differentiation* is firstly, we can take advantage of the system's ability to efficiently differentiate formulae, and more importantly, to simplify formulae. Secondly, it is aesthetically nice to be able to do all this in an integrated environment.

The first representation of a function by a DAG goes back to Kantarovich [9]. Hence this DAG is called a *Kantarovich–Graph*! There are two possible ways to traverse the graph. One method is obvious. Start at the bottom and go through all nodes to the top of the computational–tree. That's nothing else than passing the corresponding computation–sequence top–down as usual. Based on this method, the so called *forward–mode* or *forward–sweep* of *Algorithmic Differentiation* was developed [10, 3, 4].

In contrast to this *forward–mode* the *reverse–mode* or *reverse–sweep* was first used by Speelpenning [16]. He passed the underlying graph top–down and propagated the gradient backwards. He also proved that this *reverse–mode* generates automatically piece-wise optimal computation–sequences (see also [3, 4]) in the sense that all common subexpressions appeared by differentiation are optimized but not the whole computation–sequence.

In the next section we will describe the *forward–mode* and its implementation in Maple. Section 3 shows how the function GRADIENT handles local procedures. In section 4, we want to look at the syntax of GRADIENT. We summarize the class of Maple–procedures handled by it and the three subsections describe the different functions based on GRADIENT. In section 5 we draw the conclusion from this approach in Maple.

2 The forward-mode in Maple

The function GRADIENT presented in this paper is essentially an extension of the function PD [11] which is now accessible in Maple V Release 2 as the D operator which computes a single partial derivative of a function. Given a function f depending on $\vec{x} = (x_1, \ldots, x_n)$ represented by this program

Original Program

```
begin
    statement_1;
    statement_2;          (1)
        ⋮
    statement_m
end:
```

With the following notations:

- $\vec{x} = (x_1, \ldots, x_n)$ are the independent variables
- T is a set of all necessary temporary variables which depend on $\vec{x}$
- B is a set of the union of all mathematical–functions, the elementary arithmetic operations $+, -, \cdot, /$ and all possible compositions of them
- C is the set of all constants
- *statement$_i$* is a single program–statement which corresponds to one of the five types listed below:
  i) $t := h(C, T, \vec{x})$; where $t \in T, h \in B$
  ii) if–control–statement
  iii) for–statement (2)
  iv) RETURN–statement
  v) a single mathematical expression

Note:
The structure of the different branches of an if–statement and the body of a for–loop is also a statement–sequence like (1).

The source–code transformation of the original program in order to get a program for the computation of the gradient of f using the *forward–mode* isn't very difficult.

The sign ∇ denotes the gradient with respect to all variables $x_1, \ldots, x_n$. While we're passing the Kantarovich–Graph of f bottom–up, we are adding new nodes to it such that we get a larger graph representing the calculation of the gradient of f. From this graph we can derive the

Differentiated Program

```
begin
    for i from 1 to n do
        $\nabla x_i = e_i$
    od;
    ∇statement_1;          (3)
    ∇statement_2;
        ⋮
    ∇statement_m
end:
```

Where e_i denotes the i-th Cartesian basis vector in $\mathbb{R}^n$. The set G shall contain the gradients of all variables of T. All ∇*statement*$_1$,…,∇*statement*$_m$ shall be the "gradients" of the corresponding statements. Each *statement$_i$* will be replaced by ∇*statement$_i$* which is one of the following five types:

i) $\nabla t := \sum_{z \in \{x_1, \ldots, x_n\} \cup T} \frac{\partial h(C,T,\vec{x})}{\partial z} \cdot \nabla z$;
$t := h(C, T, \vec{x})$; where $t \in T, \nabla t \in G, h \in B$

ii) if–control–statement where the statement–sequences in all branches were replaced by their "gradients"

iii) for–loop where the iterated statement–sequence was replaced by its "gradient"

iv) RETURN–statement which returns the calculated gradient

v) RETURN–statement which returns the gradient of the corresponding mathematical expression

Note:

We have to compute ∇t **before** the computation of t if in the assignment $t := h(C,T,\vec{x})$, $h(C,T,\vec{x})$ itself depends on t.

The first thing we had to do was to choose a suitable data structure to represent a gradient. We decided to use the array data–type because it's very close to the mathematical formalism and allows you to build a Jacobian– or even a Hessian–procedure basing on GRADIENT quite easily. Another advantage is that the user can include the generated "gradient–procedures" in his own programs because their return types are always well defined.

Assume that $f = f(x_1,\ldots,x_n) : I\!R^n \rightarrow I\!R^m$. The corresponding procedure f shall return a one-dimensional array with m entries. By using GRADIENT we get a new procedure for ∇f that returns the Jacobian matrix of f with $m \cdot n$ entries.

Lets look at two simple examples to illustrate the use of arrays:

Example 1:

```
f := proc(x,y) local a;
   a := array(1..2);
   a[1] := x^2*y^2;
   a[2] := x^2+y^2;
   a
end:

> eval(f(x,y));

                        2  2   2    2
                     [ x  y , x  + y  ]

> df := GRADIENT(f);

df := proc(x,y)
      local da;
         da := array(1..2,1..2);
         da[1,1] := 2*x*y^2;
         da[1,2] := 2*x^2*y;
         da[2,1] := 2*x;
         da[2,2] := 2*y;
         da
      end

> eval(df(x,y));

                     [      2      2  ]
                     [ 2 x y    2 x  y ]
                     [                 ]
                     [   2 x      2 y  ]
```

Example 2:

```
> f := proc(x,y,n) local a,i;
          a := array(1..n);
          for i to n do a[i] := (x*y)^i od;
          a
       end:
> eval(f(x,y,2));
                                 2  2
                        [ x y,  x   y  ]

> df := GRADIENT(f,[x,y]);

df := proc(x,y,n)
      local a,da,i;
         da := array(1..n,1..2);
         a := array(1..n);
         for i to n do
            da[i,1] := (x*y)^i*i/x;
            da[i,2] := (x*y)^i*i/y;
            a[i] := (x*y)^i
         od;
         da
      end

> eval(df(x,y,2));

                        [    y          x    ]
                        [                    ]
                        [        2         2 ]
                        [ 2 x  y    2 x  y   ]
```

The second example shows that we even don't have to fix the array bounds! If we verify the printed results we'll see that these are just the Jacobian matrices of the underlying functions! By an extensive use of this array data–structure we don't have to care about multi valued functions. We just call the GRADIENT–function and it will create automatically the correct Jacobian matrix of f.

If we want to compute the gradient of an arbitrary mathematical expression the only thing we have to do is to differentiate the function $h(C,T,\vec{x})$ where $h \in B$ with respect to the desired variables.

The result is a vector of the form $\sum\limits_{z\in\{x_1,\ldots,x_n\}\cup T} \frac{\partial h}{\partial z} \cdot \nabla z$.

Lets look at the following example to explain the procedure.

Example:

$$h(x,y,a,b) = \sin(x)y + a^2 b$$

– x, y are the independent variables
– a, b are variables depending on x and y

Assume that da and db are declared as one-dimensional arrays with two entries denoting the partial derivatives of a and b.
If we calculate the *Total Derivative* of h we get:

$$\begin{aligned}\nabla h &= \frac{\partial h}{\partial x}\nabla x + \frac{\partial h}{\partial y}\nabla y + \frac{\partial h}{\partial a}\nabla a + \frac{\partial h}{\partial b}\nabla b \\ &= \cos(x)y \cdot \begin{pmatrix}1\\0\end{pmatrix} + \sin(x)\cdot\begin{pmatrix}0\\1\end{pmatrix} + \\ & \quad 2\,a\,b\cdot\begin{pmatrix}\frac{\partial a}{\partial x}\\ \frac{\partial a}{\partial y}\end{pmatrix} + a^2\cdot\begin{pmatrix}\frac{\partial b}{\partial x}\\ \frac{\partial b}{\partial y}\end{pmatrix} \\ &= \begin{pmatrix}\cos(x)y + 2\,a\,b\cdot da[1] + a^2\cdot db[1] \\ \sin(x) + 2\,a\,b\cdot da[2] + a^2\cdot db[2]\end{pmatrix}\end{aligned}$$

respectively.

The following example illustrates the differentiation of various kinds of statements supported by **GRADIENT**:

```
> f := proc(x,y,n) local i,t;
>         t := x*y;
>         for i to n do  t := t*x*y od;
>         if x < y then RETURN(t*x,t*y)
>         else RETURN(t,x*y)
>         fi
>      end:

> GRADIENT(f,[x,y]);

proc(x,y,n)
local dt,i,t;
   dt := array(1 .. 2);
   dt[1] := y;
   dt[2] := x;
   t := x*y;
   for i to n do
      dt[1] := dt[1]*x*y+t*y;
      dt[2] := dt[2]*x*y+t*x;
      t := t*x*y
   od;
   if x < y then
      RETURN(dt[1]*x+t,dt[2]*x,dt[1]*y,dt[2]*y+t)
   else RETURN(dt[1],dt[2],y,x)
   fi
end
```

3 Differentiation of local procedures

The differentiation of local procedures could be done in two different ways. One is, for reasons of consistency, quite obvious. We call recursively the function **GRADIENT** in order to differentiate the local procedure with respect to only one parameter. This yields a vector (gradient) of differentiated procedures. But this approach is definitely very time– and memory-consuming. A further disadvantage is that for each function call of such a component–procedure a full evaluation is unavoidable!

The other variant of differentiating local defined functions is that we immediately compute its gradient, such that we have to do only one function call to get all the derivatives we want. The run-time of the **GRADIENT**–function and the generated program will be dramatically shortened.

Important:
Local procedures will be differentiated always with respect to all their parameters. Hence it's not possible to pass an additional parameter denoting for example the dimension of an array yet. By using global defined variables this disadvantage can easily be removed. It is planed to extend the algorithm such that it should be able to detect the independent parameters of local procedures. In that case it will be possible to pass further parameters. Hence the local procedures won't be differentiated with respect to all their arguments.

For the function **GRADIENT** it's important to know the type (data structure) of the value calculated by a local function. Consider the following example

```
> f := proc(x,y) local s,d;
>      d := proc(u,v,w)
>           local a;
>              a := u^2*v*w^3; a*u*v
>           end                          ;
>      s := d(x,x*y,y);
>      s*x^2*y
> end:

> df := GRADIENT(f);

df :=

 proc(x,y)
 local d,dd,ds,lf1,s;
    ds := array(1..2);
    dd := proc(u,v,w)
          local a,da,grd;
             grd := array(1..3);
             da := array(1..3);
             da[1] := 2*u*v*w^3;
             da[2] := u^2*w^3;
             da[3] := 3*u^2*v*w^2;
             a := u^2*v*w^3;
             grd[1] := da[1]*u*v+a*v;
             grd[2] := da[2]*u*v+a*u;
             grd[3] := da[3]*u*v;
             grd
          end                          ;
    d := proc(u,v,w)
         local a;
            a := u^2*v*w^3; a*u*v
```

```
      end                           ;
   lf1 := dd(x,x*y,y);
   ds[1] := lf1[1]+lf1[2]*y;
   ds[2] := lf1[2]*x+lf1[3];
   s := d(x,x*y,y);
   RETURN(ds[1]*x^2*y+2*s*x*y,ds[2]*x^2*y+s*x^2)
 end
```

The local function **d** has three parameters and returns a single value. The differentiated function **dd** returns now the gradient as an array with three entries. This gradient is stored in the temporary variable `lf1`. In the original program the variable **s** holds the value **g(x,y,y)**, **ds** should denote now the gradient of **g(x,y,y)** with respect to **x** and **y**. The componentwise assignments to **ds[1]** and **ds[2]** lead to the correct result.

4 The GRADIENT procedure

The Maple–procedure **GRADIENT** computes ∇f where f is input as a Maple–procedure. It is possible to call the **GRADIENT** procedure in three different ways.

i) `GRADIENT(f)`

ii) `GRADIENT(f,p)`

iii) `GRADIENT(f,[x1,..,xn])`

With the following definitions:

1. **f** is a Maple procedure which computes f
2. **p** a positive integer specifying that the procedure **f** will be differentiated with respect to the first **p** parameters in the parameter list of **f**. Other parameters will be interpreted as constants.
3. **x1**, ..., **xn** are names of parameters with respect to which the gradient will be computed.

Restrictions to the Maple procedures

- it's not allowed to use **args**, **nargs** nor quotes
- recursive procedures are not supported yet
- assignments to global variables are forbidden too
- loop variables have to be constants
- only the **array** data structure is supported

What is allowed in Maple procedures

- operations with the **array** data type
- assignment statements
- **for**–loops, **while**–loops and **if**–statements
- the statements **RETURN**, **ERROR**, **break**, **print** and **lprint**
- the definition of local procedures
- the use of the elementary arithmetic operations **+**, **-**, *****, **/**, **^**
- any mathematical function Maple knows how to differentiate

4.1 The JACOBIAN procedure

The Jacobian matrix J of the function

$$\begin{array}{rccc} \vec{f}: & \mathbb{R}^n & \rightarrow & \mathbb{R}^m \\ & \vec{x} & \mapsto & \vec{y}=\vec{f}(\vec{x}) \end{array} \tag{4}$$

is defined by

$$J_{i,j}=\frac{\partial y_i}{\partial x_j} \quad \text{where} \quad \begin{array}{c} 1\leq i\leq m \\ 1\leq j\leq n \end{array} \tag{5}$$

These are naturally the entries in the Jacobian matrix $J_{\vec{f}}$ of the function $\vec{f}$.

Given a program that computes the function $\vec{f}$. Generate with the aid of *Algorithmic Differentiation* a new program for the simultaneous computation of the terms (5).

We've already shown that it's possible to generate automatically the desired Jacobian matrix by using arrays. In many scientific applications it's often the case that each component of $\vec{f}$ is given by its own function f_i $(1 \leq i \leq m)$. Thus we have m functions $f_1, \ldots, f_m$ depending on $\vec{x} = (x_1, \ldots, x_n)$ and each represented by a program. The only thing we have to do is to differentiate each function f_i and collect the results as local functions in a new procedure.

The implemented function **JACOBIAN** is based on **GRADIENT**. Thus each of the procedures **f1,..,fm** must satisfy the conditions (2). The syntax of the **JACOBIAN** procedure is similar to the **GRADIENT** procedure

i) `JACOBIAN([f1,..,fm])`

ii) `JACOBIAN([f1,..,fm],p)`

iii) `JACOBIAN([f1,..,fm],[x1,..xn])`

Where **f1,..,fm** are Maple–procedures.

Example:

```
  f1 := proc(x,y) local a;
           a := x^3*y^2;
           a*x*y
        end:
  f2 := proc(u,v) local a;
           a := 3*u^2*v^3;
           4*a*v^2
        end:


  J := JACOBIAN([f1,f2]);


J := proc(t1,t2)
       local gf,jm,i,h;
          jm := array(1 .. 2,1 .. 2);
          gf := array(1 .. 2);
          gf[1] := proc(x,y)
                      local a,da,grd;
                         grd := array(1 .. 2);
                         da := array(1 .. 2);
                         da[1] := 3*x^2*y^2;
                         da[2] := 2*x^3*y;
                         a := x^3*y^2;
                         grd[1] := da[1]*x*y+a*y;
                         grd[2] := da[2]*x*y+a*x;
                         grd
                      end                           ;
          h := gf[1](t1,t2);
          for i to 2 do  jm[1,i] := h[i] od;
          gf[2] := proc(u,v)
                      local a,da,grd;
                         grd := array(1 .. 2);
                         da := array(1 .. 2);
                         da[1] := 6*u*v^3;
                         da[2] := 9*u^2*v^2;
                         a := 3*u^2*v^3;
                         grd[1] := 4*da[1]*v^2;
                         grd[2] := 4*da[2]*v^2+8*a*v;
                         grd
                      end                                ;
          h := gf[2](t1,t2);
          for i to 2 do  jm[2,i] := h[i] od;
          jm
       end

> f1(a,b),f2(a,b);

                      4  3      2  5
                     a  b , 12 a  b


> eval(J(a,b));

                  [    3  3      4  2 ]
                  [ 4 a  b    3 a  b  ]
                  [                   ]
                  [       5      2  4 ]
                  [ 24 a b   60 a  b  ]
```

4.2 The HESSIAN procedure

The Hessian matrix H of a function $f : I\!R^n \rightarrow I\!R$ is the matrix

$$H_{i,j} = \frac{\partial^2 f}{\partial x_i \partial x_j} \quad \text{where} \quad 1 \leq i,j \leq n \tag{6}$$

Given a program that computes the function f. Generate with the aid of *Algorithmic Differentiation* a new program for the simultaneous computation of the terms (6).

The implementation of **HESSIAN** is based on **GRADIENT** too. It simply applies the **GRADIENT** function twice. For this reason the procedure f has to satisfy the conditions (2). The syntax of the **HESSIAN** procedure is exactly the same as it is for the **GRADIENT** procedure.

i) `HESSIAN(f)`

ii) `HESSIAN(f,p)`

iii) `HESSIAN(f,[x1,..,xn])`

Example:

```
> f := proc(x,y) local a; a := x^2*y; a^3*y end:
> f(x,y);
                                 6  4
                                x  y

> H := HESSIAN(f);

H :=

proc(x,y)
local a,da,da1,dda,t1;
   da1 := array(1..2);
   dda := array(1..2,1..2);
   da := array(1..2);
   dda[1,1] := 2*y;
   da[1] := 2*x*y;
   dda[2,1] := 2*x;
   da[2] := x^2;
   da1[1] := 2*x*y;
   da1[2] := x^2;
   a := x^2*y;
   t1 := array(1..2,1..2);
   t1[1,1] :=
      6*a*y*da[1]*da1[1]+3*a^2*y*dda[1,1];
   t1[1,2] := 6*a*y*da[2]*da1[1]+3*
      a^2*y*dda[2,1]+3*a^2*da1[1];
   t1[2,1] := t1[1,2];
   t1[2,2] :=
   6*a*y*da[2]*da1[2]+3*a^2*da[2]+3*a^2*da1[2];
   t1
end
```

The reader will note that the code produced by the **HESSIAN** procedure is not very cleaver. It does not realize that the arrays da and da1 are the same, and that the code could be simplified considerably. Various optimizations need to be made. Some can be made independently of the function f by the **GRADIENT**, **JACOBIAN**, and **HESSIAN** procedures. Others need to be made by a separate optimization pass. We make some more comments about this in the conclusion.

```
> eval(H(x,y));

                  [      4  4       5  3 ]
                  [ 30 x  y    24 x  y   ]
                  [                      ]
                  [      5  3       6  2 ]
                  [ 24 x  y    12 x  y   ]
```

5 Conclusion

It is natural to consider doing *Algorithmic Differentiation* in a computer algebra system because the system already knows how to differentiate and simplify formulae. In designing and implementing the **GRADIENT** routine, we have found this to be a great advantage. Our implementation extends earlier work by handling local functions. At present it does not handle recursive functions. However this is in principle not difficult and will be added shortly. We have shown that with the correct handling of arrays, Jacobians and Hessians come for free. We have not shown it here but it is not difficult to obtain the coefficients of a multivariate Taylor series as well.

In the course of our work it became apparent that the resulting procedures are not very efficient. They contain many common subexpressions, redundant statements, etc. which are naturally produced as a result of differentiating simple formulae. Perhaps more than 50% of our effort was devoted to producing more efficient code. On the one hand, we try to produce directly efficient code, i.e. independent of the function f. On the other hand, we have implemented a program optimizer that performs in a subsequent pass common sub-expression optimization and constant folding on a computation sequence, i.e. a program that consists of assignments to local variables only. There is ample room for improvement here.

The Maple programs **GRADIENT**, **JACOBIAN** and **HESSIAN** mentioned in this paper, and others, are in the Maple share library. For Maple V users, they can be obtained using anonymous ftp from neptune.inf.ethz.ch, daisy.uwaterloo.ca, or ftp.inria.fr.

References

[1] L. M. Beda et al, *Programs for Automatic Differentiation for the Machine BESM*, Inst. Precise Mechanics and Computation Techniques, Academy of Science, Moscow, 1959

[2] B. Char, K. Geddes, G. Gonnet, B. Leong, M. Monagan, S. Watt, *Maple V Language Reference Manual*, Springer, 1991

[3] A. Griewank, *On Automatic Differentiation*, Mathematical Programming, edited by M. Iri and K. Tanabe, Kluwer Academic Publishers, pp. 83-107, 1989

[4] A. Griewank, *The Chain Rule Revisited in Scientific Computing*, SIAM News May 1991, part I pp. 20-21, SIAM News July 1991, part II pp. 8-9, p. 24

[5] A. Griewank, D. Juedes, J. Srinivasan, C. Tyner, *ADOL-C, A package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. Math. Software, to appear

[6] M. Iri, *Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors – Complexity and Practicality*, Japan Journal of Applied Mathematics, Vol. 1, No. 2, pp. 223-252, 1984

[7] J. Joss, *Algorithmisches Differenzieren*, Diss. ETH 5757, 1976

[8] D. W. Juedes, *A Taxonomy of Automatic Differentiation Tools*, in A. Griewank and F. Corliss, editors, *Automatic Differentiation of algorithms: Theory, Implementation and Application*, SIAM, Philadelphia, 1991, 315–329

[9] L. V. Kantarovich, *On a mathematical symbolism convenient for performing machine calculations*, Dokl. Akad. Nauk SSSR, Vol. 113, pp. 738-741, 1957

[10] G. Kedem, *Automatic Differentiation of Computer Programs*, ACM TOMS, Vol. 6, No. 2, pp. 150-165, June 1980

[11] M. Monagan, *Program PD*, Maple share library, 1991

[12] J. Moses, *The Evolution of Algebraic Manipulation Algorithms*, IFIP Congress 1974

[13] L. B. Rall, *Automatic Differentiation – Techniques and Applications*, Springer Lecture Notes in Computer Science, Vol. 120,1981

[14] L. B. Rall, *Differentiation in PASCAL-SC : Type GRADIENT*, ACM TOMS, Vol. 10, No. 2, pp. 161-184, June 1984

[15] A. Griewank, F. Corliss, *Automatic Differentiation of algorithms: Theory, Implementation and Application*, SIAM, Philadelphia, 1991

[16] B. Speelpenning, *Compiling fast Partial Derivatives of Functions given by Algorithms*, Ph.D. Dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801

[17] R. E. Wengert, *A simple Automatic Derivative Evaluation Program*, Com. ACM, Vol. 7, pp. 463-464, 1964

[18] P. Wolfe, *Checking the Calculation of Gradients*, ACM TOMS, Vol. 6, No. 4, pp. 337-343, 1982