Maple Transactions

How to program with formulas in Maple.

MICHAEL MONAGAN, Simon Fraser University, Canada

Abstract. Maple's main strength is its ability to compute with mathematical formulas and not just with numbers. It can multiply and factor, differentiate and integrate, and simplify formulas. In this article, using differentiation as an example, I explain how to program with formulas in Maple. The key is the data representation that Maple uses for a formula and the operations Maple provides for operating on formulas. I also discuss Automatic Differentiation as an alternative which differentiates programs.

CCS Concepts: • **Computing methodologies** → **Algebraic algorithms**.

Additional Key Words and Phrases: Symbolic Differentiation, Maple Programming, Automatic Differentiation

Recommended Reference Format:
Michael Monagan. 2023. How to program with formulas in Maple.. 1, 1 (January 2023), 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

In 1979 I was taking a computing course in data structures and my instructor gave us an assignment to write a Pascal program to differentiate a formula represented by a binary tree. In the binary tree the internal nodes were the arithmetic operators and the leaf nodes were variables or integers. For example, Figure 1 shows how $x^2 + 3x + 5$ would be represented by a binary tree where I've used ↑ for the power symbol.

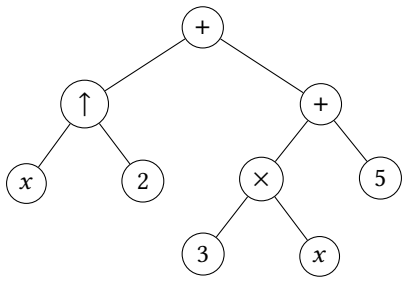

Fig. 1. A binary tree for the formula $x^2 + (3x + 5)$.

I was captivated by this assignment. What I found was that differentiating was easy because the rules for differentiation are simple, but, simplifying the derivative was difficult. My first exposure to Maple was when I took the Symbolic Computation course with Bruce Char in January 1982. Bruce showed us a Maple program to differentiate a formula. I was amazed at the brevity of the program. What I'd like to do in this article is show you how to program with formulas in Maple using the derivative problem as my working example. Programming (indefinite) integration is more difficult because the algorithm is more difficult. But, in principle, the tools that I show you for differentiation are sufficient to program integration. Now, to differentiate a formula we need

Author's address: Michael Monagan, mmonagan@sfu.ca, Simon Fraser University, Department of Mathematics, Burnaby, British Columbia, Canada, V5A-1S6.

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 an algorithm,
2 a data representation for a formula and a set of tools for operating on a formula, and
3 we need to simplify the resulting formula.

Why do we need simplification? Consider applying the sum rule to differentiate $x^2 + 3$. We differentiate each term of the sum to get $2x + 0$ then simplify this derivative to get $2x$. When we differentiate by hand we do the simplifications automatically without thinking much about them but my Pascal program had to do them explicitly. The good news is that Maple automatically simplifies formulas! So we will not have to explicitly simplify the derivative.

Since we already know how to differentiate from Calculus we need to focus on 2 the data representation. How does Maple represent a formula? What tools does Maple provide us with to work with formulas? Figure 2 shows the Maple data structure for a polynomial.

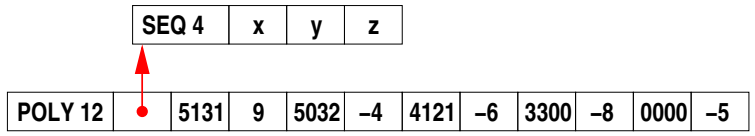

Fig. 2. Maple's data structure for the polynomial $f = 9\,xy^3z - 4\,y^3z^2 - 6\,xy^2z - 8\,x^3 - 5$.

Although I am personally very interested in the data structure, we do not need to know any details about it and we do not need to manipulate it directly. What we need is to be able to test if a given formula f in Maple is a constant, sum, product, power, function etc. so that we can apply the right differentiation rule, that is, we need to be able to "inspect" a formula. We also need to be able to "extract" parts of a formula so we can differentiate them. For example, if f is a sum of two terms, we want to extract each term of f to differentiate it. Before I show you how to do this let me say a word about Automatic Differentiation.

When you and I learned to differentiate a function $f(x)$ in Calculus, the function was a formula and the derivative was a formula. In scientific computing the function f is often represented by a computer program which contains formulas but also may have loops and subroutine calls. We use Automatic Differentiation (AD) to differentiate programs. I will end this article by showing some Maple tools for AD which differentiate a function $f : \mathbb{R}^n \to \mathbb{R}^m$ that is represented by a Maple procedure. If you haven't see AD before, it may be eye-opening.

2 Inspecting and Manipulating Maple Formulas

In the example below the formula f is a sum of two terms and the formula g is a product of three factors. We use the `type` command to inspect f, the `nops` command to tell us how many terms f has and how many factors g has and we use the `op` command to extract a term of f. Then we do likewise for the product g and function h. The Maple command `op` is short for "operand".

```
> f := 3*x+2*x^2*sin(x);
```

$$f := 3\,x + 2\,x^2\,\sin(x)$$

```
> type(f,`+`);  # f is a sum
```

$$true$$

```
> nops(f);  # f has 2 terms
```

$$2$$

```
> op(1,f);  # the first term
```

$$3x$$

```
> g := op(2,f); # the second term
```

$$2x^2 \sin(x)$$

```
> type(g,`*`);  # g is a product
```

$$true$$

```
> nops(g);  # g has 3 factors
```

$$3$$

```
> op(2,g);  # the second factor
```

$$x^2$$

```
> h := op(3,g);  # the third factor
```

$$\sin(x)$$

```
> type(h,function);  # h is a function
```

$$true$$

```
> nops(h);  #  h has one parameter
```

$$1$$

```
> op(0,h);
```

$$\sin$$

```
> op(1,h);
```

$$x$$

Our examples showed the three Maple types, '+', '*' and function. Table 1 lists all the algebraic types from which a formula is constructed. Notice that Maple does not have a '−' or '/' type. Maple views $a - b$ as $a + (-1) \times b$ and a/b as $a \times b^{-1}$.

f	type	nops(f)	op(0,f)	op(1,f)	op(2,f)
4	integer	1	Integer	4	an error
-2/3	fraction	2	Fraction	-2	3
3.14	float	2	Float	314	-2
x	symbol	1	symbol	x	an error
x[2,3]	indexed	2	x	2	3
x+y+z	`+`	3	`+`	x	y
x-y	`+`	2	`+`	x	$-y$
-x*y	`*`	3	`*`	-1	x
x/y	`*`	2	`*`	x	y^{-1}
x^n	`^`	2	`^`	x	n
sin(x)	function	1	sin	x	an error
J(v,y)	function	2	J	v	y

Table 1. Maple's algebraic types and their operands

We will use type to inspect a Maple formula and op and nops to extract operands of a formula. These three Maple commands are sufficient to write any program in Maple that computes with a formula. But to simplify our code I will also use the typematch command and the map command. To create a new formula we just use the arithmetic operators + - * / ^ and functions. Before we get into the rules for differentiation consider the following Maple examples

```
> diff(x^2,x);
```

$$2x$$

```
> diff(f(x),x);
```

$$\frac{d}{dx} f(x)$$

```
> lprint(diff(f(x),x):
diff(f(x),x):
```

In the first example the Maple diff command computed the derivative $2x$. In the second example diff could not do anything as $f(x)$ is an unknown function. But it did return something. It returned itself as you can see from the output of lprint. In Maple lingo we say diff *returned unevaluated*. How can we write a Maple procedure that returns unevaluated? We do it this way.

```
DIFF := proc(f::algebraic,x::name)
    if f=x then 1 else 'DIFF'(f,x) end if;
end proc;
```

Procedure DIFF takes as input two arguments, f the expression to be differentiated and x the variable of differentiation. Procedure DIFF only knows one differentiation rule, namely, $dx/dx = 1$. Otherwise it returns unevaluated. The quotes around DIFF tell Maple to not execute DIFF recursively, which would result in an infinite recursion. Instead Maple creates and returns the unevaluated function call DIFF(f,x). Note, the Maple type algebraic is for a formula; it includes all the types listed in Table 1. Note, the Maple type name is for a variable; it is either a symbol or indexed type (see Table 1). Now we can add the rules for differentiation.

3 Programming Differentiation

Let us begin by implementing the following rules for differentiation.

Rule 1: if f is a constant the derivative is 0 e.g. DIFF(2/3,x)=0.

Rule 2: if f is a variable then the derivative is 1 if $f = x$ otherwise 0, e.g. DIFF(y,x)=0.

There are three numerical types in Maple, namely, integers, fractions and decimal numbers. They are of type integer, fraction, and float respectively. See Table 1. So rule [1] could be encoded this way

```
if type(f,integer) or type(f,fraction) or type(f,float) then 0
```

The Maple type numeric is short for any one of integer, fraction or float so the type test can be simplified to type(f,numeric). For rule 2 we use the Maple type name which includes symbols (a letter followed by one or more letters or digits) e.g. x, x12 and alpha, and indexed (subscripted) variables e.g. x[1] and A[1,2]. This means we have

```
DIFF := proc(f::algebraic,x::name)
    if type(f,numeric) then 0
    elif type(f,name) then
        if f=x then 1 else 0 end if
    else 'DIFF'(f,x)
    end if;
end proc;
```

We now add the sum rule, the product rule and the power rule.

Rule 3 (Sum rule): $\frac{d(u+v)}{dx} = \frac{du}{dx} + \frac{dv}{dx}$.

Rule 4 (Product rule): $\frac{d(u \times v)}{dx} = \frac{du}{dx}v + u\frac{dv}{dx}$.

Rule 5 (Power rule): if n is a constant then $\frac{d(u^n)}{dx} = n\frac{du}{dx}\,u^{n-1}$.

One way to code to the sum rule is to use a for loop as follows. We use `nops(f)` to tell us how many terms are in the sum and `op(i,f)` to extract the i'th term.

```
elif type(f,`+`) then
    d := 0;
    for i to nops(f) do
        d := d + DIFF(op(i,f),x)
    end do;
    d;
```

This loop builds up the derivative one term at a time which is not efficient when `nops(f)` is very large. The loop can be replaced with either of the following equivalents

```
d := add( DIFF(op(i,f),x), i=1..nops(f) );
d := add( DIFF(u,x), u in f );
```

When I teach Calculus (our university uses Stewart's text) I only teach our students how to differentiate a product of two factors $u \times v$. If we have more than two factors, $u \times v \times w$ for example, then we would apply the product rule twice using

$$\frac{d(u(vw))}{dx} = \frac{du}{dx}(vw) + u\frac{d(vw)}{dx} = \frac{du}{dx}(vw) + u(\frac{dv}{dx}w + v\frac{dw}{dx})$$

I will use the general product rule for $f = \prod_{i=1}^{n} f_i$ which is

$$\frac{df}{dx} = \sum_{i=1}^{n} d\frac{f_i}{dx}\,\frac{f}{f_i}.$$

Here is a nice way to code this:

```
elif type(f,`*`) then
    add( DIFF(op(i,f),x)*(f/op(i,f)), i=1..nops(f) );
```

Here is a really nice way:

```
elif type(f,`*`) then
    add( DIFF(u,x)*(f/u), u in f )
```

The power rule $d\,f^n/dx = n\frac{df}{dx}f^{n-1}$ is only true if n is a constant. How will we test if a Maple formula n is a constant? The test `type(n,numeric)` is inadequate as it would not handle y or $\sin(1)$ or $\int_1^2 f(t)dt$. The right way to test for a constant, mathematically speaking, is to test if the derivative of n is 0, that is, if `DIFF(n,x)=0` ! So we have

```
if type(f,`^`) and DIFF(op(2,f),x)=0 then
    u := op(1,f); n := op(2,f);
    n*DIFF(u,x)*u^(n-1);
```

This is a little clumsy. I'll use Maple's `typematch` command which avoids the two explicit assignments of u and n. The above code is equivalent to the following.

```
if typematch(f,(u::algebraic)^(n::algebraic)) and DIFF(n,x)=0 then
    n*DIFF(u,x)*u^(n-1)
```

The `typematch` command first tests if f is of type `algebraic^algebraic`. If it is `typematch` then assigns the variables u and n to the corresponding operands of f and returns `true`. If not `typematch` returns `false`. Assembling rules 1 through 5 we have the following code.

```
DIFF := proc(f::algebraic,x::name)  local i,u,n;
    if type(f,numeric) then 0
    elif type(f,name) then
        if f=x then 1 else 0 end if
    elif type(f,`+`) then
        add(DIFF(u,x), u in f)
    elif type(f,`*`) then
        add(DIFF(u,x)*(f/u), u in f)
    elif typematch(f,(u::algebraic)^(n::algebraic)) and DIFF(n,x)=0 then
        n*DIFF(u,x)*u^(n-1)
    else 'DIFF'(f,x)
    end if;
end proc;
```

This is already a powerful routine. And so concise! Here is an example

```
> f := 3*x^2+x*exp(x)+3;
```

$$f := 3\,x^2 + x\,\mathrm{e}^x + 3$$

```
> DIFF(f,x);
```

$$6x + \mathrm{e}^x + DIFF(\mathrm{e}^x, x)x$$

What we are missing is the derivative of functions and also how to differentiate derivatives and antiderivatives. Let's add differentiation rules for exponentials and logarithms.

Rule 6: $d\dfrac{e^u}{dx} = \dfrac{du}{dx}e^u$ and

Rule 7: $d\dfrac{\ln(u)}{dx} = \dfrac{du}{dx}\dfrac{1}{u}$.

The Maple representations for e^x and $\ln x$ are `exp(x)` and `ln(x)` respectively. Notice that e^x is represented as a function in Maple, not a power. This code will work

```
elif type(f,function) and op(0,f)=exp then
    u := op(1,f);
    DIFF(u,x)*exp(u);
elif type(f,function) and op(0,f)=ln then
    u := op(1,f);
    DIFF(u,x)/u;
```

Again, using typematch, we can simplify these to

```
elif type(f,'exp'(u::algebraic)) then DIFF(u,x)*exp(u)
elif type(f,'ln'(u::algebraic)) then DIFF(u,x)/u
```

The reason we put the ' quotes here is to avoid the cost of evaluating the exponential and the logarithm. Notice that if $f = \ln 2$ the differentiation rule for ln will result in $du/dx = 0$ so `DIFF(ln(2),x)` will output 0. We now give two further rules from integral calculus.

Rule 8 (Antiderivative): $\frac{d}{dx}\int f(x)dx = f(x)$ and $\frac{d}{dx}\int f(x,y)dy = \int(\frac{d}{dx}f(x,y))dy$.

Rule 9 (Definite integral) $\frac{d}{dx}\int_a^b f(t)dt = \frac{db}{dx}f(b) - \frac{da}{dx}f(a)$.

Rules 8 and 9 are easy using `typematch`. We just need to know that the antiderivative $\int f(x,y)dx$ is represented in Maple as the Maple function `int(f(x,y),x)` and the definite integral $\int_a^b f(t)dt$ is represented as `int(f(t),t=a..b)`. Here are the rules.

```
 elif typematch(f,'int'(g::algebraic,y::name)) then
     if y=x then g else int(diff(g,x),y) end if
elif typematch(f,'int'(g::algebraic,y::name=a::algebraic..b::algebraic)) then
     DIFF(b,x)*eval(g,y=a)-DIFF(a,x)*eval(g,y=b)
```

The representation of a definite integral in Maple uses the equation type and the range type. Let us illustrate how we can work with these types using the `type`, `nops` and `op` commands.

```
> h := int(f(t),t=0..a);
```

$$\int_0^a f(t)dt$$

```
> nops(h);
```

$$2$$

```
> op(0,h);
```

$$int$$

```
> op(1,h); # the integrand
```

$$f(t)$$

```
> e := op(2,h);
```

$$e := t = 0..a$$

```
> type(e,equation);
```
$$true$$
```
> r := op(2,e);
```
$$r := 0..a$$
```
> type(r,range);
```
$$true$$

Here is our most difficult rule to code.

Rule 10 (Partial derivatives commute): $\dfrac{\partial^2}{\partial y \partial x} f(x,y) = \dfrac{\partial^2}{\partial x \partial y} f(x,y)$.

Our Maple representation of the two partial derivatives is

`DIFF(DIFF(f(x,y),x),y)` and `DIFF(DIFF(f(x,y),y),x)`.

Our idea is to compare the two variables x and y when we see a derivative in our DIFF procedure and sort the variables. The rule we want to program is

if $x < y$ then apply `DIFF(DIFF(f(x,y),y),x)` $\longrightarrow$ `DIFF(DIFF(f(x,y),x),y)`.

We can use Maple's set ordering to decide if $x < y$ or not. When you create a set in Maple e.g.

```
> S := {z,y,x,z};
```
$$S := \{x, y, z\}$$

Maple sorts the elements of a set and removes duplicates. For symbols it sorts them in alphabetical order. So we could use

```
elif typematch(f::'DIFF'(g::algebraic,y::name)) and x<>y then
    if op(1,{x,y}) = x then DIFF(DIFF(g,x),y) else f end if;
```

There is another way to do this. The Maple command `lexorder(x,y)` returns true if the symbol x is less than or equal to y in lexicographical order. For example

```
> lexorder(y,x), lexorder(x,y), lexorder(x,x);
```
$$false,\ true,\ true$$

So an alternative is

```
elif typematch(f::'DIFF'(g::algebraic,y::name)) and x<>y
    and lexorder(x,y) then DIFF(DIFF(g,x),y)
```

Does rule 10 work for nested cases like

`DIFF(DIFF(DIFF(f(x,y,z),x),y),z) - DIFF(DIFF(DIFF(f(x,y,z),z),y),x)`

which should simplify to 0? It turns out that it does! It sorts the second derivative to be the same as the first. Can you see what sorting algorithm rule 10 is using?

Finally, we consider the Maple types sets, lists, Vectors and Matrices which are collections of objects where we would like to simply differentiate each entry of the set, list, Vector, or Matrix.

Rule 11: if f is a set, list, or rtable then differentiate each entry of f.

Note type rtable includes the types Array, Matrix and Vector. Instead of using a loop we will use the map command as follows.

```
elif type(f,list) then map(DIFF,f,x)
```

The command map(F,L) applies the Maple function or procedure F to the "components" of L. The command map(F,L,X) also passes the additional argument X to F as illustrated below.

```
> L := [ y, 2*x, sin(x) ];
```

$$L := [y,\ 2\,x,\ \sin(x)]$$

```
> map(F,L);
```

$$[F(y),\ F(2\,x),\ F(\sin(x))]$$

```
> map(F,L,x);
```

$$[F(y,x),\ F(2\,x,x),\ F(\sin(x),x)]$$

To include sets and rtables as well as lists we use

```
elif type(f,{list,set,rtable}) then map(DIFF,f,x);
```

In map(F,A), map does not always apply F to the operands of A. The exceptions are the atomic objects, tables and rtables. I've put the details for the map command in an Appendix. Here is our final DIFF program.

```
DIFF := proc(f::{algebraic,list,set,rtable},x::name)
  local i,u,v,n,y,g,a,b;
  if type(f,numeric) then 0
  elif type(f,name) then # DIFF(x,x)=1 and DIFF(y,x)=0
     if f=x then 1 else 0 end if
  elif type(f,`+`) then # sum rule
     add( DIFF(u,x), u in f )
  elif type(f,`*`) then # product rule
     add( DIFF(u,x)*(f/u), u in f )
  elif typematch(f, (u::anything)^(n::anything) ) and DIFF(n,x)=0 then
     n*DIFF(u,x)*u^(n-1) # power rule
  elif typematch(f,'exp'(u::anything)) then DIFF(u,x)*exp(u)
  elif typematch(f,'ln'(u::anything)) then DIFF(u,x)/u
  elif typematch(f,'int'(g::algebraic,y::name)) then
     if y=x then g else int(diff(g,x),y) end if;
 elif typematch(f,'int'(g::algebraic,y::name=a::algebraic..b::algebraic)) then
     DIFF(b,x)*eval(g,y=a)-DIFF(a,x)*eval(g,y=b)
 elif typematch(f,'DIFF'(u::anything,y::name)) and x<>y and lexorder(x,y) then
     DIFF(DIFF(u,x),y) # sort partial derivative when x<y
  elif type(f,{list,set,rtable}) then map(DIFF,f,x)
  else 'DIFF'(f,x)
  end if
end proc;
```

4 Automatic Differentiation

All of us learned how to differentiate a formula. The derivative of a formula is another formula which we get by applying the rules of differentiation. That's what Maple's `diff` command does. Here is an example.

```
> f := sin(x+y)*cos(x+y)*exp(-x-y);
```

$$f := \sin(x+y)\cos(x+y)\mathrm{e}^{-x-y}$$

```
> g := [diff(f,x),diff(f,y)];
```

$$g := [\,\left(\cos^2(x+y)\right)\mathrm{e}^{-x-y} - \left(\sin^2(x+y)\right)\mathrm{e}^{-x-y} - \sin(x+y)\cos(x+y)\,\mathrm{e}^{-x-y},$$

$$\left(\cos^2(x+y)\right)\mathrm{e}^{-x-y} - \left(\sin^2(x+y)\right)\mathrm{e}^{-x-y} - \sin(x+y)\cos(x+y)\,\mathrm{e}^{-x-y}\,]$$

Notice that the size of both partial derivatives is significantly larger than the size of the f. This is because of the product rule. If we wanted to use the formula for f or g in a program, we would write f and g as functions perhaps in C or Fortran or Matlab. Here is f as a Maple procedure.

```
> F := proc(x,y)
>   local t,s,c,e,r;
>   t := x+y;
>   s := sin(t);
>   c := cos(t);
>   e := exp(-t);
>   r := s*c*e;
> end proc;
```

Automatic Differentiation (AD for short) takes as input a program and outputs a program which computes the derivatives. We compute $\frac{\partial F}{\partial x}$ using the D operator. Here `D[1]` means differentiate F with respect to the first argument.

```
> D[1](F);
```

```
proc(x, y)
local c, cx, e, ex, s, sx, t;
   t := x + y;
   sx := cos(t);
   s := sin(t);
   cx := -sin(t);
   c := cos(t);
   ex := -exp(-t);
   e := exp(-t);
   c*e*sx + c*ex*s + cx*e*s
end proc
```

Notice that the original statements in F appear in the derivative (except for the last statement `r := s*c*e` which is not needed) and each statement is preceded by it's derivative (except for the first `tx := 1;` which has been optimized away). Notice also that the derivative has repeated common subexpressions. We can optimize these away as follows

```
> with(codegen):
> optimize(D[1](F));
```

```
proc(x, y)
local c, e, s, sx, t, t1;
   t := x + y;
   sx := cos(t);
   s := sin(t);
   c := sx;
   t1 := exp(-t);
   e := t1;
   c*e*sx - c*s*t1 - e*s^2
end proc
```

Maple's `optimize` command is a Maple program that extracts the statements and formulas from inside a Maple procedure, identifies common subexpressions using Maple's builtin hash tables, removes them, and constructs a new Maple procedure. Here is the gradient of F.

```
> with(codegen):
> G := GRADIENT(F,mode=forward);
```

```
G := proc(x, y)
local c, dc, de, ds, dt, e, r, s, t;
   dc := array(1 .. 2);
   de := array(1 .. 2);
   ds := array(1 .. 2);
   dt := array(1 .. 2);
   dt[1] := 1;
   dt[2] := 1;
   t := x + y;
   ds[1] := dt[1]*cos(t);
   ds[2] := dt[2]*cos(t);
   s := sin(t);
   dc[1] := -dt[1]*sin(t);
   dc[2] := -dt[2]*sin(t);
   c := cos(t);
   de[1] := -dt[1]*exp(-t);
   de[2] := -dt[2]*exp(-t);
   e := exp(-t);
   return c*e*ds[1] + c*s*de[1] + e*s*dc[1],
       c*e*ds[2] + c*s*de[2] + e*s*dc[2]
end proc
```

The procedure G was constructed using the "forward mode" of AD. For each statement it computes the partial derivatives with respect to x and y and puts them in an array. In the following we compute the gradient using the "reverse mode' of AD.

```
> H := GRADIENT(F,mode=reverse);
```

```
H := proc(x, y)
local c, df, e, s, t;
   t := x + y;
   s := sin(t);
   c := cos(t);
   e := exp(-t);
   df := array(1 .. 4);
   df[4] := c*s;
   df[3] := s*e;
   df[2] := e*c;
   df[1] := -df[4]*exp(-t) - df[3]*sin(t) + df[2]*cos(t);
   return df[1], df[1]
end proc
```

It will be less obvious how the reverse mode works. Notice that the first four statements in H are copied from F. The last statement `r := s*c*e;` is omitted because it is not needed. The reverse mode applies the chain rule starting from the last statement `r := s*c*e;` in procedure F and works backwards through the statements in F. It first computes the partial derivatives of r with respect to the local variables s, c and e appearing in r and not x and y. For example $df[4] = \frac{\partial r}{\partial e} = cs$.

As you can see the size of the procedure computed with the reverse mode is smaller than the one obtained with the forward mode. One of the most important results in AD is that if F is a function of N variables $x_1, x_2, \ldots, x_n$ with M arithmetic operations the number of arithmetic operations in the gradient using the forward mode is $O(MN)$ but in the reverse mode is only $O(M+N)$! Details of the two modes of AD are presented on the Wikipedia page for Automatic Differentiation. Let's optimize H.

```
> H := optimize(H);
```

```
H := proc(x, y)
local c, df, e, s, t, t7;
   t := x + y;
   s := sin(t);
   c := cos(t);
   e := exp(-t);
   df := array(1 .. 4);
   df[4] := c*s;
   df[3] := s*e;
   df[2] := e*c;
   df[1] := c*df[2] - e*df[4] - s*df[3];
   t7 := df[1];
   return t7, t7
end proc
```

Hmm, how do we know H is correct? Let's check it.

```
> eval(g,{x=1.3,2.4}), H(1.3,2.4);
```

$$[-0.00026718023, -0.00026718023], [-0.000267180242, -0.000267180242]$$

That is convincing but it is not a proof. Here is a proof.

```
> [H(x,y)]-g;
```

$$[0, 0]$$

Here H(x, y) builds the formulas for the partial derivatives. One of the editors challenged me as to whether this really constitutes a proof since by my use of the word "proof", he expected to see a formal argument or an application of a program like the Coq proof assistant (see see https://coq.inria.fr) that formally proves theorems. I replied that all forms of "proofs", both human written proofs and computer generated proofs are prone to error; just as Maple's simplifier is. In this case I trust Maple's simplifier because it has been used millions of times and an error on a simple example like this is now highly unlikely.

5 Appendix

In `map(F,L)`, the map command generally applies F to the operands of L as defined by the op command. For example if $L = [1, 2, 3]$ then `map(F,L)` map outputs the list $[F(1), F(2), F(3)]$ and if $L = x + y + z$ then `map(F,L)` map outputs $F(x) + F(y) + F(z)$. The exceptions are the atomic types, the table types and the series type.

Atomic types. The atomic types are integer, fraction, numeric, complex(extended_numeric), symbol, indexed, string procedure, and 'module'. If A is an atomic type then `map(F,A)` returns $F(A)$. For example, `map(F,2/3)` returns $F(2/3)$. It does not return $F(2)/F(3)$.

Table types. The table types are table and rtable where rtable includes the Array, Matrix, and Vector types. For a table type T, `map(F,T)` applies the function F to the entries of the table and not the operands. Here is an example.

```
> A := Array(1..3,[5,6,7]);
```

$$A := [5,\ 6,\ 7]$$

```
> type(A,Array), type(A,rtable), type(A,Vector);
```

$$true,\ true,\ false$$

```
> map(F,A);
```

$$[F(5),\ F(6),\ F(7)]$$

The series type. For a series S, `map(F,S)` applies F to the coefficients of the series S, and not the terms of S. For example

```
> S := taylor(ln(1+x),x);
```

$$S := x - \frac{1}{6}x^3 + \frac{1}{120}x^5 + O\left(x^7\right)$$

```
> map(F,S);
```

$$F\left(1\right)x + F\left(-\frac{1}{6}\right)x^3 + F\left(\frac{1}{120}\right)x^5 + F\left(O\left(1\right)\right)x^7$$

```
> type(S,series), nops(S), op(0,S);
```

$$true,\ 8,\ x$$

```
> seq( op(i,S), i=1..8 ); # the operands of S
```

$$1, 1, -1/6, 3, 1/120, 5, O(1), 7$$

As an example of an application of the atomic type and the map command we give a recursive Maple procedure to evaluate a formula at $x = a$. Here `EVAL(f,x,a)` is equivalent to Maple's `eval(f,x=a)`.

```
> EVAL := proc(f::anything,x::name,a::algebraic)
>    lprint(f); # just to see what happens
>    if f=x then a
>    elif type(f,atomic) then f
>    else map(EVAL,f,x,a)
>    end if;
> end proc:
>
> L := [x^2+5/7,y=J(v,x),int(f(t),t=a..x)]:
> EVAL(L,x,z);
[x^2+5/7, int(f(t),t = a .. x)]
x^2+5/7
x^2
x
2
5/7
int(f(t),t = a .. x)
f(t)
t
t = a .. x
t
a .. x
a
x
```

$$\left[z^2 + \frac{5}{7}, \int_a^z f(t)\,dt\right]$$