Differential algebras without differentials: an easy C++ implementation.

LEO MICHELOTTI
Fermilab*, P.O.Box 500, Batavia, IL 60510

1 Introduction.

In the field of real numbers, R, differentiation is an analytic operation requiring the evaluation of a limit. In non-Archimedean field extensions of R, such as Robinson's [9,5] $^*R$, differentiation may be definable as an arithmetic operation. Rall [8,7] demonstrated the practicality of this observation by implementing the procedure on a computer. Any problem combining differentiation with numerical analysis seems ripe for this idea; applications run the gamut from solving functions with Newton's method to calculating Lyapunov exponents for the orbits of a dynamical system. Berz has introduced the method into accelerator applications and has heroically written a FORTRAN pre-compiler, DA, which can be used to differentiate automatically the nonlinear mappings associated with tracking programs and thereby construct their polynomial approximations.[2,3] Forest, Berz, and Irwin already have shown how this capability can be used to find normal forms for the Hamiltonians which these mappings represent.[6]

In view of the importance of this technique, it is unfortunate that some descriptions of the basic ideas still leave a number of readers confused as to why it actually works. The confusion which arises usually is connected either with employing the notion of "differential" or with the connection to $^*R$, and its implied machinery of ultrafilters. In fact, automated differentiation can be motivated and explained rather plainly without any reference to infinitesimals or differentials whatsoever. We shall describe one possible approach in this paper.

The method which we shall use will suggest its own implementation. However, FORTRAN is not the most natural language in which to carry it out. In the second section we shall describe an almost trivial implementation using C++ [10] (Indeed, one of the motivations for writing this paper is to persuade militant FORTRAN extremists to invest the four or five days necessary to learn this powerful and easy language.) Take heed, however, that what we describe below is only a stripped-down implementation, written in three days, of differential algebra's most essential features; it is not as robust as and does not contain the battery of tools available in Berz's DA package, the product of a significant amount of work.

2 Prolongation structures, pro-numbers, and differential algebras.

With any smooth function, $f : R^n \to R$, we associate its "prolongation structure," or "prolongation" for short.[1]

$$\hat{f} = \langle f,\ \nabla f,\ \nabla\nabla f,\ \nabla\nabla\nabla f,\ \ldots\rangle\ .$$

The brackets $\langle\ldots\rangle$ indicate an ordered set of objects of different kinds, as opposed to the components of a vector, which are objects of the same kind. The first member of $\hat{f}$ is the function f itself, the second is its gradient, the third is its hessian, and so forth. Evaluating a prolongation at some value of its argument — say, $\hat{a} = \hat{f}(\boldsymbol{x})$ — yields a new structure which we shall call, for lack of anything better, a "pro-number."

$$\hat{a} = \hat{f}(\boldsymbol{x}) = \langle f(\boldsymbol{x}),\ \nabla f(\boldsymbol{x}),\ \nabla\nabla f(\boldsymbol{x}),\ \nabla\nabla\nabla f(\boldsymbol{x}),\ \ldots\rangle = \langle a, \underline{a}, \underline{\underline{a}}, \ldots\rangle$$

The first member of $\hat{a}$ is a real number, its second member is a singly indexed array of reals, its third is a doubly indexed array of reals, and so forth. We shall call the number of indices associated with a member its "order."

Now, let the symbol $\otimes$ represent a binary operation on smooth functions: addition, multiplication, convolution, or whatever. We can extend its domain of definition to include prolongations in a natural manner.

$$\hat{f}\otimes\hat{g} \equiv \widehat{f\otimes g} = \langle f\otimes g,\ \nabla(f\otimes g),\ \nabla\nabla(f\otimes g),\ \ldots\rangle$$

This definition assures that information about the derivatives of functions propagates correctly through the operations. The simplest of these is addition. Let f and g be two functions with prolongations $\hat{f}$ and $\hat{g}$. Then we define the sum, $\hat{f}+\hat{g}$, as follows.

$$\hat{f}+\hat{g} \equiv \widehat{f+g} = \langle f+g,\ \nabla f+\nabla g,\ \nabla\nabla f+\nabla\nabla g,\ \ldots\rangle$$

Development of product and quotient operations is a little more complicated but follows straightforwardly from repeated application of the chain rule.

$$\begin{aligned}
\nabla(fg) &= g\nabla f + f\nabla g\\
\nabla\nabla(fg) &= g(\nabla\nabla f) + f(\nabla\nabla g) + (\nabla g)(\nabla f) + (\nabla f)(\nabla g)\\
&\ \vdots
\end{aligned}$$

$$\begin{aligned}
\nabla(f/g) &= (1/g)\nabla f - (f/g^2)\nabla g\\
\nabla\nabla(f/g) &= (1/g)\nabla\nabla f - (f/g^2)\nabla\nabla g\\
&\quad (1/g^2)((\nabla f)(\nabla g) + (\nabla g)(\nabla f)) + (2f/g^3)(\nabla g)(\nabla g)\\
&\ \vdots
\end{aligned}$$

Upon evaluation, these analytic identities become rules for doing arithmetic with pro-numbers. Let $\hat{a} = \hat{f}(\boldsymbol{x}) = \langle a, \underline{a}, \underline{\underline{a}}, \ldots\rangle$ and $\hat{b} = \hat{g}(\boldsymbol{x}) = \langle b, \underline{b}, \underline{\underline{b}}, \ldots\rangle$ be two arbirary pro-numbers. The rules for constructing their sum, $\hat{s} = \hat{a}+\hat{b}$, product $\hat{p} = \hat{a}\hat{b}$, and quotient, $\hat{q} = \hat{a}/\hat{b}$ are obtained directly from the above identities.

$$\begin{array}{lll}
s = a + b, & \underline{s} = \underline{a} + \underline{b}, & \underline{\underline{s}} = \underline{\underline{a}} + \underline{\underline{b}},\ \ \ldots\\
p = ab, & \underline{p} = a\underline{b} + b\underline{a}, & \underline{\underline{p}} = b\underline{\underline{a}} + a\underline{\underline{b}} + \underline{a}\,\underline{b} + \underline{b}\,\underline{a},\ \ \ldots\\
q = a/b, & \underline{q} = (1/b)\underline{a} - (a/b^2)\underline{b}, & \underline{\underline{q}} = (1/b)\underline{\underline{a}} - (1/b^2)(\underline{a}\,\underline{b} + \underline{b}\,\underline{a})\\
 & & \quad +(2a/b^3)\underline{b}\,\underline{b} - (a/b^2)\underline{\underline{b}},\ \ \ldots
\end{array} \tag{1}$$

A few observations:
(a) Pro-numbers along with their arithmetic operations form an algebra which obeys the same commutative, associative, and distributive laws as real numbers. There is a "zero" and "unit" of this algebra: if we define $\hat{1} = \langle 1, \underline{0}, \underline{\underline{0}}, \ldots\rangle$ and $\hat{0} = \langle 0, \underline{0}, \underline{\underline{0}}, \ldots\rangle$, then it is easy to confirm that for any $\hat{a}$, $\hat{a}+\hat{0} = \hat{a}$ and $\hat{1}\cdot\hat{a} = \hat{a}$. Unlike reals, however, the algebra is not an integral domain, much less a field: not all non-zero elements can be inverted: the inverse of any pro-number whose first member is (the real number) zero is not defined. Its exact algebraic classification is a commutative ring with identity.
(b) For a pro-number to be interpretable as the evaluation of a prolongation, all its multi-indexed members should be symmetric under permutation of their indices. However, it is possible to enlarge the set of pro-numbers by admitting non-symmetric members which obey the same arithmetic rules. Since permutation symmetry is preserved by the arithmetic operations, the symmetric pro-numbers form an invariant subalgebra of this larger pro-number algebra.
(c) A nice test of division is to note that for all invertible $\hat{a}$, $\hat{a}/\hat{a} = \langle 1, \underline{0}, \underline{\underline{0}}, \ldots\rangle = \hat{1}$.

Of course, any implementation of this algebra must be truncated. In light of this, an important feature of the arithmetic rules is that the m^{th} member of the result of an operation depends only on members of the operands in positions $\leq m$. For example, $\underline{p}$ depends on a, b, $\underline{a}$, and $\underline{b}$, but not on $\underline{\underline{a}}$, $\underline{\underline{b}}$, or any other members of higher order. This means that if we truncate the pro-number algebra at various orders we obtain new algebras, the "differential algebras," that obey the same commutative, associative, and distributive laws as the original. (These are related to the ${}_nD_v$ algebras in Berz.[3])

Suppose we truncate at the first order (second member), so that $\hat{a} = \langle a, \underline{a}\rangle$ is now a generic element of the algebra. If $a = 0$, then, by the rule for multiplication, $\hat{a}^2 = \langle 0, \underline{0}\rangle = \hat{0}$. Thus, the truncated algebra possesses elements whose square is (the truncated) zero. This is one of the fundamental properties of an "infinitesimal." If we truncate at the second order, then

*Operated by the Universities Research Association, Inc. under contract with the U.S. Department of Energy.

[1] This object may have a more accepted name; if so, I do not know what it is. My calling it a prolongation is based on an abuse of terminology introduced on page 3 of reference [1]. The word already has several meanings, depending on the mathematical context, so I do not consider it very harmful to add one more.

$\hat{a} = \langle 0, 0, \underline{\underline{a}} \rangle$ would be the form of a differential element. However, an element of the form $\hat{a} = \langle 0, \underline{a}, \underline{\underline{a}} \rangle$, with $\underline{a} \neq 0$, would satisfy $\hat{a}^3 = \hat{0}$ while $\hat{a}^2 \neq \hat{0}$, which is not at all the same thing. If we truncate the prolongation algebra at order m, then any element, $\hat{a}$, whose first member is zero will satisfy $\hat{a}^{m+1} = \hat{0}$. Rather than calling these "infinitesimals," we shall refer to them as "nilpotent" elements.[2] There are no nilpotent elements in the full prolongation algebra, with its infinity of members; they appear when we truncate operations at some finite order.

By repeated use of the arithmetic operations, we can define rational functions of prolongations. These can be extended to transcendental functions — $\cos(\hat{u})$, $\exp(\hat{u})$, and so forth — either using a power series, as in reference [3], or by the following approach. Let $u : R^n \to R$ and $g : R \to R$ be functions, with different domains, and consider their concatenation, $h(\boldsymbol{x}) = g(u(\boldsymbol{x}))$. Since $h : R^n \to R$, its domain is appropriate for the same dimension of prolongation as u. We shall extend the domain of g by defining $g^{\#}(\hat{u}) \equiv \hat{h}$. (In principle, it should not be necessary to invent a new symbol for this extended function; examining the type of the argument should resolve ambiguities. Nonetheless, we shall let $g^{\#}$ represent extended g to avoid confusion.) Its members are obtained by using chain rule once again.

$$\begin{aligned} \nabla[g(u)] &= g'(u) \nabla u \\ \nabla \nabla [g(u)] &= g''(u) (\nabla u)(\nabla u) + g'(u) \nabla \nabla u \\ &\vdots \end{aligned}$$

As with the arithmetic operations, upon evaluation these rules translate into definitions for transcendental functions of pro-numbers. For any $\hat{a}$ we have $g^{\#}(\hat{a}) = \hat{b}$, where

$$b = g(a), \quad \underline{b} = g'(a)\,\underline{a}, \quad \underline{\underline{b}} = g''(a)\,\underline{a}\,\underline{a} + g'(a)\,\underline{\underline{a}}, \quad \ldots$$

For example, for $g = \cos$ we have,

$$\cos^{\#}(\hat{a}) = \langle \cos(a), \; -\sin(a)\,\underline{a}, \; -\cos(a)\,\underline{a}\,\underline{a} - \sin(a)\,\underline{\underline{a}}, \; \ldots \rangle \quad , \tag{2}$$

while for $g = \exp$ we would have,

$$\exp^{\#}(\hat{a}) = \langle e^a, \; e^a\,\underline{a}, \; e^a(\underline{a}\,\underline{a} + \underline{\underline{a}}), \; \ldots \rangle \quad .$$

A special set of prolongations are obtained from projections onto the coordinates. We shall denote these by $\hat{x}_k$; their pro-number evaluations are given by

$$\hat{x}_k(\boldsymbol{x}) \equiv \langle x_k, \underline{e}_k, \underline{\underline{0}}, \ldots \rangle \quad , \tag{3}$$

where the components of $\underline{e}_k$ are $(\underline{e}_k)_i = \delta_{ik}$, and all members of $\hat{x}_k$ vanish beyond the first order. Notice that the second member is indeed the gradient of the associated projector. These special elements generalize the notion of "variable" and are just what is required to initialize a numerical computation of derivatives, as we shall see below.

3 An implementation in C++.

The importance of the pro-number algebra and analysis, or its truncated counterparts, is that they provide a mechanism for propagating derivatives through arithmetic operations, which are the most complicated things that computers can actually do with numbers, and function calls. In this way, differentiation becomes an arithmetic, rather than analytic, procedure, thereby rendering it appropriate for automated computation. Suppose, for example, that one wants to evaluate the derivatives of

$$g(x, y, z, t) = \cos(2x^2y^3) + 2\exp(zt/x) - 3y^2;$$

at some particular values of its arguments, say at $x = -1.54$, $y = -1.17$, $z = 0.35$, and $t = 1.59$. Rather than analytically writing these derivatives and then evaluating or numerically taking finite differences to approximate differentiation we could proceed by using "differential" algebra variables. The calculation is initialized by setting

$$\hat{x} = \langle -1.54, \underline{e}_1, \underline{\underline{0}}, \ldots \rangle, \; \hat{y} = \langle -1.17, \underline{e}_2, \underline{\underline{0}}, \ldots \rangle, \tag{4}$$

$$\hat{z} = \langle 0.35, \underline{e}_3, \underline{\underline{0}}, \ldots \rangle, \; \hat{t} = \langle 1.59, \underline{e}_4, \underline{\underline{0}}, \ldots \rangle \tag{5}$$

and proceeds by evaluating $g^{\#}(\hat{x}, \hat{y}, \hat{z}, \hat{t})$, in the process of which we obtain not only the value of the function but its derivatives as well, since they are propagated through the computation. These are then obtained by reading the members of the result.

Imagine a programming language which contains pro-numbers as a type of variable — like reals, integers, or complex variables — and whose compiler recognizes arithmetic operations on these variables as well as provides the standard transcendental functions. A program for carrying out the above computation might contain statements like the following:[3]

```
nstd x, y, z, t, g;           // Illustrative
                              // program
x.setVariable( -1.54, 0 );    // segment.
t.setVariable(  1.59, 1 );
y.setVariable( -1.17, 2 );
z.setVariable(  0.35, 3 );

g = cos( 2.0*(x*x)*(y*y*y) )
    + 2.0*exp( z*t/x ) - 3.0*y*y;
```

The first line declares the variables to be pro-numbers; the name of the declared type is **nstd** , for "non-standard."[4] The **setVariable** statements which follow simply initialize these variables to their desired values and tell the program the index which is to be associated with each; they implement the assignments shown in Eq.(5). (The ordering is arbitrary; I have made t follow x just to be perverse. It is convenient to begin indices in C and C++ at 0 rather than 1, but this is not a necessary restriction.) The final statement will evaluate not only the function but its derivatives as well.

Now, no languages possess **nstd** type variables as part of their basic structure. However, some modern, more powerful languages, like C++ or Ada, allow one effectively to enlarge the language by defining new "classes" of variables. Loosely speaking, a class is created in C++ by specifying the data peculiar to it and the functions and operations which can access those data. In particular, the C++ class **nstd** is partially defined by the following user interface:

```
overload cos;                                  // nstd
overload exp;                                  // Interface

class nstd {
  double f;                                    // Line 5
  double df[ DIMENSION ];
  double ddf[ DIMENSION ][ DIMENSION ];
public:
  nstd();
  nstd( double );                              // Line 10
  void setVariable( double, int );
  friend nstd operator+( nstd&, nstd& );
  friend nstd operator-( nstd&, nstd& );
  friend nstd operator-( nstd& );
  friend nstd operator*( nstd&, nstd& );      // Line 15
  friend nstd operator/( nstd&, nstd& );
  friend nstd cos( nstd );
  friend nstd exp( nstd );
} ;
```

To save space, I've truncated at the second order, included only two transcendental functions, and otherwise shortened the class definition. However, enough remains to illustrate the ideas behind the implementation. Lines 5-7 describe the data appropriate to a **nstd** variable: a double precision real, vector, and matrix — all of which correspond to saying that $\hat{a} = \langle a, \underline{a}, \underline{\underline{a}} \rangle$.[5] The size of the arrays is contained in a macro variable, **DIMENSION**, which is defined by the user prior to compilation. These lines comprise the "private" part of the declaration; the "public" part that follows lists the functions and operators which are allowed to access these data.

When the C++ compiler comes to a **nstd** variable declaration in an application program it will allocate enough memory to accomodate the data declared in the class definition. Further, it will initialize these data according to rules specified in the "constructor" function declared in Line 9. The constructor which I use simply initializes everything to zero.[6]

[2] We have not introduced an ordering within which the nilpotents are "small," as is done in the fields $^*R$ of Robinson or **No** of Conway.[9,5,4] Berz has removed this deficiency by defining a lexicographic ordering scheme, and shown that the connection with true differentials proceeds as you would expect. (However, the resulting ordered algebra is still not a field. Nor do theorems proven in the algebra automatically translate into theorems in R, as is the case with *R.)

[3] I hope these program segments are transparent enough to make this section readable without C++ expertise. Nonetheless, those who have used C++ undoubtedly will understand the examples better.

[4] This designation is probably a mistake, in light of my reluctance to call the nilpotent elements differentials.

[5] This is easily extended; arrays in C++ can have an arbitrarily large number of indices.

[6] For the record, **FORALL** is not a part of the C++ language; it is a macro I predefined to make subsequent programming easier and more transparent.

```
nstd::nstd() {
int i,j;
f = 0.0;
FORALL(i) {
  df[i] = 0.0;
  FORALL(j) ddf[i][j] = 0.0;
  }
}
```

Starting a calculation is done with the **setVariable** (public) member function, which implements the operation shown in Eq.(3) and Eq.(5).

```
void nstd::setVariable( double x, int j ) {
int i,k;
f = x;
FORALL(i) {
  df[i] = 0.0;
  FORALL(k) ddf[i][k] = 0.0;
  }
df[j] = 1.0;
}
```

Binary operations, such as multiplication or addition, are defined for these classes by "overloading" the arithmetic symbols, which is accomplished by defining corresponding member functions for the class. For example, the member function **operator***, which is declared in Line 15 and is meant to implement multiplication, is written (truncated at second order) as follows.

```
nstd operator*( nstd& x, nstd& y ) {
int i,j;
nstd z;
z.f = x.f * y.f;
FORALL(i) {
  z.df[i] = ( x.f * y.df[i] ) + ( y.f * x.df[i] );
  FORALL(j)
    z.ddf[i][j] = ( y.f * x.ddf[i][j] )
                + ( x.f * y.ddf[i][j] )
                + ( x.df[i] * y.df[j] )
                + ( y.df[i] * x.df[j] ) ;
  }
return z;
}
```

The **operator/** function is written similarly. By comparing with Eq.(1) we see that these functions are nothing more than a literal translation of the arithmetic rules. There is no need for table lookups or explicit computations of memory locations; C++ takes care of those details automatically.

Of course, the applications programmer neither needs nor wants to know how **operator***, **nstd::setVariable**, **nstd::nstd** or any other member function is implemented. He only cares that if **x**, **y**, and **z** are declared to be **nstd** variables in his program, then a statement like **z = x*y** will be interpreted correctly and perform as expected. Only the interface is required to do that; the "toolkit" which contains the implementation is completely transparent, buried in a library with which he will eventually link his programs. Further, if the interface is contained in a file, say **nstd.hxx**, then the applications program need only contain the lines

```
#define DIMENSION 4    // or 2 or 6 or whatever
#include "nstd.hxx"
```

to allow the use of **nstd** variables.

By itself, **operator*** specifies the rule for multiplying two **nstd** variables, but in the "Illustrative program segment" written above we multiplied **nstd** variables by constants, 2.0 and 3.0. This worked properly because of the second constructor function, declared in Line 10 of the interface. Because the class contains the member **nstd(double)** the C++ compiler will automatically convert any real (i.e., type **double**) constant or variable appearing in an expression with **nstd** variables to its **nstd** counterpart. This avoids the necessity (1) of writing separate multiplication routines for all possible combinations of variables, **nstd operator*(double&, nstd&)**, and the like, or (2) of declaring all variables in a program as type **nstd**. The constructor is implemented as one would expect,

```
nstd::nstd( double x ) {
int i,j;
f = x;
FORALL(i) {
  df[i] = 0.0;
  FORALL(j) ddf[i][j] = 0.0;
  }
}
```

but, once again, these details are transparent to the user.

Finally, since C++ does type checking, we can overload functions as well as operators. This allows us, for example, to define a **nstd** member function called "cos" which performs exactly as specified in Eq.(2).

```
nstd cos( nstd x ) {
double sn,cs;
int i,j;
nstd w;
sn = sin( x.f );
cs = cos( x.f );
w.f = cs;
FORALL(i) {
  w.df[i] = - sn*x.df[i];
  FORALL(j)
    w.ddf[i][j] = - cs*x.df[i]*x.df[j] - sn*x.ddf[i][j];
  }
return w;
}
```

The other elementary transcendental functions are extended in a similar manner.

And that really is all there is to it. The result is that **nstd** variables act as though they were part of the original language: the compiler knows how to initialize them, arithmetic operations use the same symbols, transcendental functions are called by their same names, and type conversion takes place automatically in mixed variable expressions. Of course, there is no reason to stop; one can go on to develop toolkits for **rational**, **polynomial**, **lorentzGroup**, **beamLine**, **observable**, **quaternion**, **extensionField**, or any other mathematical object which may be useful. Using C++, or other object-oriented languages, even changes the approach to problem solving. Rather than immediately asking, "How do I write the program?" one first steps back and asks, "What objects are most convenient for expressing the problem and obtaining a solution?" Toolkits created to aid the solution of one problem are then easily reusable and appear naturally in later programs. Ideally, given some level of communication among a group of users, toolkits can be shared thereby multiplying the group's productivity. In short, the arrival of object-oriented programming (which is what this amounts to) represents a significant breakthrough of almost limitless possibilities.

References

[1] Robert L. Anderson and Nail H. Ibragimov. *Lie-Bäcklund Transformations in Applications.* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1979. SIAM Studies in Applied Mathematics.

[2] Martin Berz. *Nuclear Instruments and Methods*, A258:431, 1987.

[3] Martin Berz. Differential algebraic description of beam dynamics to very high orders. *Particle Accelerators*, 24(2), February 1989. to be published.

[4] John H. Conway. *On Numbers and Games.* Academic Press, New York, 1976.

[5] Martin Davis. *Applied Nonstandard Analysis.* John Wiley & Sons, New York, 1977.

[6] Etienne Forest, Martin Berz, and John Irwin. Normal form methods for complicated periodic systems: a complete solution using differential algebra and lie operators. *Particle Accelerators*, 24(2), February 1989. to be published.

[7] L. B. Rall. The arithmetic of differentiation. *Mathematics Magazine*, 59:275-282, 1986.

[8] L. B. Rall. Automatic differentiation: techniques and applications. In *Lecture Notes in Computer Science No. 120*, Springer-Verlag, 1981.

[9] Abraham Robinson. *Non-Standard Analysis, Studies in Logic and the Foundations of Mathematics.* North-Holland, 1966.

[10] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.