Fermi National Accelerator Laboratory

受入
90-5-24 1
高工研図書室

FN-535

MXYZPTLK: A Practical, User-Friendly C++ Implementation of Differential Algebra: User's Guide

Leo Michelotti
Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, Illinois 60510

January 31, 1990

Operated by Universities Research Association Inc. under contract with the United States Department of Energy

MXYZPTLK: A Practical, User-Friendly C++ Implementation of Differential Algebra: User's Guide

Leo Michelotti

1 Introduction

Except for Section 3, which is intended to motivate as well as to inform, this note is written for people who already (a) have heard about differential algebra and want to use it and (b) write programs in C++ . Those who have not yet been exposed to these topics are invited to read the references in the bibliography. Here we shall only describe how to use Version 1.0 of MXYZPTLK, a differential algebra toolkit[1] programmed using GNU C++ on a SUN workstation. The toolkit implements two classes, DA and DAVector, which model the numbers of differential algebra, just as "double" variables model real numbers. As its name suggests, a DAVector contains, as *part* of its structure, an array of DA variables; it is introduced primarily for concatenation.

In a previous paper[7] I described a C++ class, **nstd**, which directly and very straightforwardly modelled "prolonged" numbers, an object which we shall review briefly below, as a way of doing differential arithmetic. This concept, which merely extended Rall's [11] pedagogical methods to higher dimensions, provided an easily grasped construct for understanding how the most basic operations of differential algebra work. I had hoped also that it would motivate people to learn more about object-oriented programming, especially as it is done in C++ . However, the **nstd** class was thrown together in a few weeks, and although it could do simple calculations on low dimensional spaces, it was *not* (and was never intended to be) a practical implementation for more sophisticated problems: (a) it used too much memory, much of which accomplished nothing more than storing zeroes; (b) the arithmetic rules spent much of their time operating on these useless zeroes; (c) worst of all, the maximum "weight" of an **nstd** variable — the highest order of derivatives stored — was specified in the class definition rather than by the application program; (d) most of the more powerful, and correspondingly more useful, differential algebra operations, such as concatenation and differentiation, were awkward to implement.

MXYZPTLK corrects these deficiencies. The classes DA and DAVector model prolonged numbers as dynamically allocated (and deallocated) doubly linked lists — which we shall call simply "chains" — whose attributes are defined at runtime. Uninformative zeroes are neither stored nor operated upon, the maximum weight stored is specified by the user, and all differential algebra operations are implemented easily. Each link in a chain contains the "index array" (to be defined below) for a particular *non-zero* derivative and its value. When it is first declared, the variable is "null," a chain with no links. Its links are created

[1]Or tool; I have difficulty distinguishing between the two.

dynamically as calculations proceed in the application program. Runtime attributes are specified using a **DASetup** function, after which arithmetic and analytic operations proceed in a transparent manner.

This transparency is important; MXYZPTLK has been designed for "user-friendliness"; this User's Guide should provide enough information to start using the DA and DAVector classes immediately. The syntax and notation is as close to "expected" as I could make it, and each variable keeps track of such attributes as its accuracy or its "reference point" (see below), freeing the application program from routine bookkeeping tasks. DA and DAVector variables are declared in the usual way. Once declared, the components of a DAVector may be accessed just as though it were an array, as demonstrated in the following program fragment.

```
...
DA x, y, z;
DAVector u;
...
x = u[0];
y = u[1];
z = u[2];
```

However, a statement like "`u[1] = y`" is not allowed; this type of assignment is handled by a **.setVariable** method, to be described below. Notice in the fragment that **u** is *not* declared with the statement, "`DA u[3]`." Such a declaration would provide for an array of DA variables but would not specify one of class DAVector.

The rest of this note describes how to use these classes in C++ programs. Before beginning, however, we shall review a few pieces of jargon, some of which we have used already. With any smooth function, $f : U \subseteq R^N \rightarrow R$, we associate its "prolongation," which explicitly stores information on derivatives of the function.[2]

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

The first member of this structure, its "standard part," is the function f itself, the second is its gradient, the third is its hessian, and so forth. Everything except the standard part is the "nilpotent"[3] part; it is also called the "differential" or "infinitesimal" or "nonstandard" part. Evaluating a prolonged function at a "reference point" — say, $\hat{a} = \hat{f}(\underline{w})$ — produces a "prolonged number."

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

The **nstd** class described in [7] modelled this (truncated) structure exactly; the chains of the DA class provide greater efficiency. The underlying subset, $U \subseteq R^N$, which is the domain of our prolonged functions is the "problem space." As determined by the application, it splits into two "sectors" which comprise the "dynamical" and "control" coordinates, and accordingly, $N = N_d + N_c$. (For example, suppose that we are studying the restricted three-body problem: say, the motion of a small satellite under the influence of a planet and its moon. The dynamical sector would represent the six-dimensional phase space, corresponding to the initial momentum and position of the satellite. However, if we wanted to examine such questions

[2] My using this term is an abuse of terminology introduced on page 3 of reference [1]. The word already has several meanings, depending on context; this just adds one more.

[3] So-called because under the rules of differential arithmetic, a truncated prolongation whose standard part is zero eventually vanishes under repeated multiplication.

as the sensitivity of the final state to the masses of the planet and moon, then we would add these as "control" coordinates of the problem space.) The "index array" associated with a derivative is the ordered array of integers which specify the derivative. (For example, if the problem space is 3 dimensional, say $\underline{x} = (x_0, x_1, x_2)$, then the index array associated with $\partial^6 f(\underline{x})/\partial x_0 \partial x_1^3 \partial x_2^2$ would be $(1, 3, 2)$.) The sum of the indices (the components of an index array) will be varyingly referred to as the derivative's "weight," its "order," or its "degree."

2 Functions and methods

In this section we describe the functions and methods[4] currently available in MXYZPTLK, arranged in the order in which they probably would be used in most programs.

2.1 Setup functions

```
(a) void DASetup( int n, int w )
(b)      DASetup( int n, int w, double* r )
(c)      DASetup( int n, int w, int s )
(d)      DASetup( int n, int w, int s, double* r )
```

Before DA variables can be used, the application program must provide information on the dimensions of the problem space and on an initial reference point. This is done by four **Setup** functions, one of which must be invoked before using DA variables in arithmetic or analytic operations. The formal arguments, all input, are interpreted as follows.

int n: Dimension of the problem space, the total number of dynamical and control coordinates.

int w: The maximum derivative weight to be carried by DA variables. If we interpret a DA variable as a multinomial, then its degree will be $\leq$ **w**.

int s: The number of dynamical coordinates, i.e., the dimension of the dynamical sector, or "phase space."

double r[n]: An array containing the reference point.

If form (a) or (b) is used, so that s is not declared explicitly, the default option sets $s = 0$. In practice this means that all variables are considered to be control variables, and neither concatenation nor Poisson brackets will be allowed (see Sections 2.7 and 2.8). If form (a) or (c) is used, so that the reference point is not declared, it is set to the "origin," an array of zeroes. When form (c) or (d) is used, so that s is specified, **DASetup** will stop the application program if its arguments s and **n** do not satisfy $1 \leq \mathbf{s} \leq \mathbf{n}$.

In principle, **DASetup** should be invoked before the formal declaration of DA variables, but this is not always possible. For example, an application program may contain a fragment like this:

[4] A "method" is a public member function of either the DA or DAVector class.

```
DA x;
DA y;

main() {
DASetup();
...
```

Here, x and y are meant to be global variables, so they are initialized when the program begins to run and *before* the **DASetup** function can be invoked. What happens in such a case is this: the C++ DA constructors only *partially* initialize these variables and load their addresses into a queue. When **DASetup** is finally invoked, this queue is traversed, and the initialization of any variable which had been declared previously is completed.

2.2 Setting the reference point

(a) void **DAFixReference(double* r)**
(b) **DAFixReference(DA& x)**
(c) **DAFixReferenceAtStart(DAVector& x)**
(d) **DAFixReferenceAtEnd(DAVector& x)**
(e) void **DA::fixReference()**
(f) **::fixReference(double* r)**
(g) **::fixReference(DA& x)**
(h) **::fixReferenceAtStart(DAVector& x)**
(i) **::fixReferenceAtEnd(DAVector& x)**
(j) void **DAVector::fixReference()**
(k) **::fixReference(double* r)**
(l) **::fixReference(DA& x)**
(m) **::fixReferenceAtStart(DAVector& x)**
(n) **::fixReferenceAtEnd(DAVector& x)**

A default reference point is established initially by a **DASetup** fuction; variables declared either before or after its invocation are assigned this reference point as their own. However, the default reference point may be changed by one of the first four functions. The first sets it value to that of an array provided by the user. Changing this array later in the application program will not, by itself, change the default reference; another invocation of **DAFixReference** would be required. Form (b) of this function sets (or resets) the default reference point to that of an already defined DA variable. The third function, **DAFixReferenceAtStart**, sets the default reference to the reference point of its argument; the fourth, **DAFixReferenceAtEnd**, sets it to the ***standard part*** of its argument. For example, suppose the first component of a DAVector u prolongs the function $\cos(xy + \pi/2)$, while the second component prolongs $\sin(xy + \pi/2)$, both about the point $(x, y) = (\sqrt{\pi}, -\sqrt{\pi})$. Then "`DAFixReferenceAtStart( u )`" would set the default reference to $(\sqrt{\pi}, -\sqrt{\pi})$, while "`DAFixRefernceAtEnd( u )`" would set it to $(0, -1)$. The latter function is essential for doing concatenation correctly (see Sections 2.7 and 3.3).

The ten methods (e)-(n) are public members of the DA and DAVector classes. They perform analagously to the first four, but rather than acting on the default reference point, these members adjust the reference point of the individual variables. For example, in the fragment

```
DA x, y, z;
...
x.fixReference( y );
z.fixReference();
```

the **.fixReference** member sets the reference point of **x** to that of y, while the reference point of **z** is set to the current default reference.

2.3 Initializing a calculation: coordinates

void DA::setVariable(int j)
::setVariable(double x, int j)
void DAVector::setVariable(DA& x, int j)
::setVariable(double x, int j)

DA variables model prolonged numbers. Arithmetic must begin by identifying a set of variables as prolonged coordinates. The best way of doing this is first to set the default reference point with **DASetup** or **DAFixReference**. Then one simply assigns an "index" to each coordinate variable, as in the fragment below.

```
...
double r[3];

r[0] =  0.0;
r[1] =  1.0;
r[2] = -1.0;

DASetup( 3, 12, r );

DA x, y, z, f;

x.setVariable( 0 );
y.setVariable( 1 );
z.setVariable( 2 );

f = exp( x*y + z );
```

This identifies the phase space coordinate array, $\underline{u} \equiv (x, y, z)$. The variable **f** will contain data on the prolonged function, $f(\underline{u}) = e^{xy+z}$, with derivatives evaluated at the point $\underline{u} = (0, 1, -1)$. These data can be accessed through a selection method (explained in Section 2.6) by using the indices that were assigned by

.setVariable.

A second way of initializing a DA calculation employs the second form of **.setVariable** to declare a DA variable as a coordinate while simultaneously setting its value. This method is not recommended: it resets the default reference point one component at a time, so that a invoking **.fixReference** would be required after the fact.

```
...
DASetup( 3, 12 );
DA x, y, z, f;

x.setVariable( 0.0, 0 );
y.setVariable( 1.0, 1 );
z.setVariable( -1.0, 2 );

x.fixReference();
y.fixReference();

f = exp( x*y + z );
```

The two DAVector methods enable one to declare a component of a DAVector variable to be a coordinate — which is useful in the control sector — or to load DA variables into specific components — prior to concatenation, for example. Their use will be illustrated in Section 3.

2.4 Operators

Logical and arithmetic binary operators act the way one naturally expects. The replacement operator, =, enables the replacement of one DA, or DAVector, variable by another, while the logical operators == and != test whether two variables are equivalent. Arithmetic operators +, -, * and /, when sandwiched between two DA variables, activate the corresponding arithmetic operations of addition, subtraction, multiplication, and division. In addition, the subtraction symbol, -, also acts as a unary operator on DA variables, indicating that they are to be negated. The C++ operators +=, -=, *=, and /= are available as well.

When placed between two DAVector variables, the "multiplication" operator, *, initiates concatenation rather than multiplication. This will be discussed in detail in Section 2.7.

Components of a DAVector can be accessed by the postfix unary operator, [ ]. For example,

```
DA x, y, z;
DAVector u;
...
x = u[0];
y = u[1];
z = u[2];
...
```

will load the zeroth component of **u** into x, the first into y, and so forth.

In addition to these, the binary operator caret, ^ , placed between two DAVectors performs a Poisson bracket. We delay its description to Section 2.8.

All binary operators *except concatenation*, which has its own subtleties, check to be sure that their two operands have the same reference point. If they do not, then an error message is written on the standard output, and the application program is stopped. Of course, the replacement operator, =, automatically sets the reference point of its left-hand operand to that of the right-hand one.

2.5 Transcendental functions

Ideally, all C++ transcendental functions available for "`double`" variables should be available for DA variables as well. However, the only ones written for Version 1.0 are the exponential and circular functions, **cos, cosh, exp, sin, sinh, tan, tanh.** Each takes a single DA variable as its argument and returns a single DA variable as its result. The C++ functions most conspicuously missing are **pow** and **log**. These will be made available in Version 1.1, and more will be added to subsequent versions of this toolkit.

2.6 Selection methods

```
double DA::standardPart()
           ::derivative( int* m )
           ::weightedDerivative( int* m )
DA     DA::filter( int wgtLo, int wgtHi )
void   DAVector::standardPart( double* x )
                 ::derivative( int* m, double* x )
                 ::weightedDerivative( int* m, double* x )
```

A number of methods access parts of DA variables without changing the variable. As a DA method, **.standardPart**, returns as its value the standard part of the variable; as a DAVector method, it accepts an array pointer (that is, the name of an array) argument and loads the standard parts of all its components into this array. For example:

```
...
double z, x[8];
DAVector y;
...
y.standartPart( x );
z = y[3].standartPart();
if( z == x[3] ) cout << "All is OK\n"
...
```

The **.derivative** and **.weightedDerivative** routines return the value of a specified derivative. The argument is a pointer to (name of) an integer array containing the indices of the desired derivative. For example, if `f` represents the DA evaluation of a prolonged function of three variables, f, at the point $\underline{w}$, then the derivative $\partial^6 f(\underline{x})/\partial x_0 \partial x_1^3 \partial x_2^2|_{\underline{x}=\underline{w}}$ can be obtained as follows.

```
...
DA f;
double d, w[3];
int m[3];
...
DASetup( 3, 10, w );
...
m[0] = 1;
m[1] = 3;
m[2] = 2;
d = f.derivative( m );
...
```

The **.weightedDerivative** returns the derivative weighted by factorials of the indices. These are the actual coefficients which would appear in a multinomial representation of f, and they, not the derivatives, are the actual numbers stored in the DA variable.[5] Thus, if we replace `.derivative` with `.weightedDerivative` in the example above, then the value returned would be $(1!\,3!\,2!)^{-1}\partial^6 f(\underline{x})/\partial x_0 \partial x_1^3 \partial x_2^2|_{\underline{x}=\underline{w}}$.

As with **.standardPart**, the DAVector methods load the values of the derivative at each component into the array pointed to by the additional argument, **double* x**.

The **.filter** method, which currently belongs only to the DA class, returns a DA variable whose links store those derivatives with weights bounded by the arguments, **wgtLo** and **wgtHi**. It was written primarily for use by other DA methods, and I do not expect it to be used much in application programs.

2.7 Evaluation and concatenation

double DA::multiEval(double* x)
DAVector operator*(DAVector& x, DAVector& y)

DA variables can act like power series. The data stored in the links of a DA variable are the coefficients of a power series, or multinomial, expansion of a function, f, about the reference point. This expansion can be evaluated by using the **.multiEval** method.

Closely related to evaluation is the operation of concatenation, which is activated by sandwiching the "multiplication" binary operator, $*$, between two DAVectors. Let $\underline{f}, \underline{g} : R^{N_d+N_c} \rightarrow R^{N_d+N_c}$ be two mappings of the problem space into itself *which act like the identity on the control sector*. That is, only the dynamical coordinates change under the action of $\underline{f}$ and $\underline{g}$; the control variables are not touched. The composite map, $\underline{h} = \underline{f} \circ \underline{g} : \underline{u} \mapsto \underline{f}(\underline{g}(\underline{u}))$, is a mapping of the same type and, therefore, also representable by a DAVector. However, notice that although the reference points of $\underline{h}$ and $\underline{g}$ are identical, say $\underline{a}$, the reference point of $\underline{f}$ is $\underline{g}(\underline{a})$.

When the control sector is not empty, *all DAVector operations and methods assume that the first N_d components refer to the dynamical sector and the final N_c to the control sector.*

[5] In fact, the **.derivative** method first invokes **.weightedDerivative** and then multiplies by the factorials.

Further description of these methods would be awkward without recourse to examples; we shall continue this discussion in Section 3.

2.8 Differentiation and Poisson brackets

DA DA::D(int* m)
DA operator^ (DA& x, DA& y)

Derivatives of functions are themselves functions, and DA variables contain a method, **.D**, which implements this operation. For example, if u and v are functions over R^2, and we want a functional correspondence, $v \equiv \partial^3 u/\partial x_0^2 \partial x_1$, this would be accomplished as follows.

```
...
DA u, v;
int m[2];
...
m[0] = 2;
m[1] = 1;
v = u.D( m );
```

The DA variable u itself is unchanged by this method.

Taking derivatives *lowers the maximum accurate weight* of a DA variable. Thus, if u stores derivatives of $u(\underline{x})$ through weight w, and we define v to be an m^{th}-order derivative of u, then v can store the derivatives of v accurately only through weight $w - m$, all derivatives of higher weight being unknown. One of the private members of each DA variable keeps track of the maximum weight of derivatives computed accurately, which may be less than the maximum weight declared by the **DASetup** function. These numbers are propagated through arithmetic operations, so errors will not arise when using less accurate variables. In principle, an applications program can request a differentiation or invoke a selection method which cannot be carried out accurately. If this happens, then the DA class will stop the program and write an error message to the standard output.

If the dynamical sector has even dimension, say $N_d = 2n$, it can be (and usually is) interpreted as a phase space whose first n components are "positions" and whose second are "momenta." The Poisson bracket of two such observables is an observable, and DA implements this operation via the binary operator ^ .

```
...
DA f, g, h;
...
h = f^g;
```

Because this operation requires taking derivatives, the accurate weight of the resultant is smaller than that of its operands, and just as with **.D**, if the operation cannot be carried out the program will stop.

2.9 Miscellaneous utilities

void DAAbout()

void DANews()
void DA::peekAt()
::clear()
void DAVector::peekAt()

The first function, **DAAbout**, prints information about the DA implementation, especially the version number, on the standard output; the second, **DANews** prints notices of differences between one implementation and another, as well as information on known bugs.

The **.peekAt** method will traverse a DA variable and write information on each link to the standard output, including the index array of the associated derivative, its value, and the address of the link. It provides a way of looking at the entire variable without repeated use of the **.derivative** methods described in Section 2.6. The method **.clear** returns a DA variable to its null state, a chain with no links. It should be used rarely and cautiously.

3 Examples

We display below a few sample programs which illustrate various features of the DA toolkit. They are simple enough to be understandable without detailed knowledge of C++ , and I hope that, to some extent, their ease and transparency will help motivate those who continue to hesitate investing the four or five days needed to learn this language or who have been discouraged by essays [13] suggesting that FORTRAN still possesses a few advantages.

3.1 Poisson brackets

This little test program evaluates the Poisson bracket of two functions and tests DA against the Jacobi identity in a four dimensional phase space. The first Poisson bracket is carried out on the functions

$$\begin{aligned} a(\underline{x},\underline{p}) &= x_1^2 x_2^3 p_1 p_2^4 \ , \\ b(\underline{x},\underline{p}) &= \sin(x_1 p_2^2 x_2^3) \ . \end{aligned}$$

The bracket is to be evaluated at the arbitrarily selected point, $(\underline{x},\underline{p}) = (0.32, 0.5, -3.1, 1.5)$. For testing the Jacobi identity, we introduce a third function, $c = \exp(p_1 x_1 + p_2 x_2)$.

```
#include <std.h>
#include <stdio.h>
#include <stream.h>
#include "da.hxx"               // ** SEE COMMENT 1

main() {
```

```
double u1, u2, v1, v2;
double r[4];
r[0] = u1 = 0.32;
r[1] = u2 = 0.5;
r[2] = v1 = -3.1;
r[3] = v2 = 1.5;

DASetup( 4, 3, 4, r );          // ** SEE COMMENT 2

double w, y, z, answer;
DA a, b, c;
DA x1, x2, p1, p2;
DA pb;

x1.setVariable(0);              // ** SEE COMMENT 3
x2.setVariable(1);
p1.setVariable(2);
p2.setVariable(3);

                                // ** SEE COMMENT 4
a = (x1*x1) * (x2*x2*x2) * p1 * (p2*p2*p2*p2);
b = sin( x1 * (p2*p2) * (x2*x2*x2) );
pb = a^b;
cout << "Computed by DA: " << pb.standardPart() << "\n";

w = (u1*u1) * (u2*u2*u2) * v1 * (v2*v2*v2*v2);
y =      u1 * (v2*v2) * (u2*u2*u2)  ;
z = cos( y );
answer = w*y*z*( 6.0/(u2*v2) - 1.0/(u1*v1) - 12.0/(u2*v2) );
cout << "Exact answer  : " << answer << "\n";

cout << "And also      : "     // ** SEE COMMENT 5
     << ( ( (x1*x1) * (x2*x2*x2) * p1 * (p2*p2*p2*p2) ) ^
          ( sin( x1 * (p2*p2) * (x2*x2*x2) ) )
        ).standardPart()
     << "\n\n";

// -- Test of the Jacobi identity
c = exp( p1*x1 + p2*x2 );
pb = (a^(b^c)) + (b^(c^a))
               + (c^(a^b));     // ** SEE COMMENT 6
cout << "Jacobi identity\n";
pb.peekAt();                    // ** SEE COMMENT 7
}
```

Output:
```
Computed by DA: .125897
Exact answer  : .125897
And also      : .125897

Jacobi identity

Count  = 35, Weight = 3, Max accurate weight = 1
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00
Weight: 0   Value: 2.498002e-16   || Addresses: 313360 184296 184560 : 184248
Index:  0  0  0  0

Weight: 1   Value: 7.771561e-16   || Addresses: 184296 184560 193136 : 184512
Index:  0  0  0  1

Weight: 1   Value: 1.110223e-16   || Addresses: 184560 193136 191528 : 184584
Index:  0  0  1  0

Weight: 1   Value: -1.332268e-15   || Addresses: 193136 191528 191648 : 193160
Index:  0  1  0  0

Weight: 1   Value: -3.774758e-15   || Addresses: 191528 191648 190592 : 191552
Index:  1  0  0  0
```

Comment 1: The DA header file, da.hxx, must be included at the top of any application program. This form of the statement assumes, of course, that it is located in the same directory as the program.

Comment 2: We assume a four dimensional phase space with no control sector. Only terms up through weight 3 are to be kept. The default reference is set at a rather unusual location just for illustrative purposes. We choose to invoke **DASetup** before declaring DA variables, but as explained in Section 2.1, this is not a restriction.

Comment 3: These four statements initialize the calculation by establishing coordinates.

Comment 4: The Poisson bracket is computed two ways: (1) here, using the binary operator ^ on DA variables **a** and **b** and (2) below, for comparison, using its algebraic expansion on variables of type double.

Comment 5: This third calculation emphasizes that DA methods and operators work not only on formally declared DA variables *but also on expressions which evaluate to DA variables*. It is carried out without introducing auxiliary variables, such as **a**, **b**, or **pb**.

Comment 6: The extra parentheses make certain that everything gets evaluated in the proper order. Not only is the Poisson bracket operation non-associative (and non-commutative), its precedence relative to addition is ambiguous.

Comment 7: The output from this invocation of **.peekAt** shows that **pb** contains 35 links. Only those whose weights are not above the maximum accurate weight are displayed here. The fact that the rest continue to be stored is a flaw (but not a bug) in the toolkit which will be removed from Version 1.1. By the Jacobi identity, all links of pb should store a zero, which means that they should not be present, since zeroes are not stored beyond the standard part. However, because of numerical roundoff errors, the actual values stored are on the order of 10^{-15}. A second improvement in the toolkit will be to drop all results of addition or subtraction which are on the order of machine accuracy and to deallocate the corresponding links.

3.2 Evaluation

Now we calculate e using two different power series.

```
#include <std.h>
#include <stdio.h>
#include <stream.h>
#include "da.hxx"

main() {

cout << "The correct result is: " << exp( 1.0 ) << "\n\n"

double r[3];
r[0] = 0.5;
r[1] = 0.4;
r[2] = 0.0;
DASetup( 3, 7, r );            // ** SEE COMMENT 1

DA x, y, z;
DA u, v;

x.setVariable(0);
y.setVariable(1);
z.setVariable(2);

u = exp( x );
v = exp( x + y + z );

double s[3];
r[0] = 1.0;                    // ** SEE COMMENT 2
r[1] = 0.0;
r[2] = 0.0;
s[0] = 0.33;
s[1] = 0.33;
s[2] = 1.0 - s[0] - s[1];
for( int w = 1; w <= 7; w++ )  // ** SEE COMMENT 3
  cout << w << ":     " << (u.filter( 0, w )).multiEval( r )
            << "     " << (v.filter( 0, w )).multiEval( s )
            << "\n";
}
```

Output:

```
The correct result is: 2.71828

1:    2.47308    2.70556
2:    2.67917    2.71786
3:    2.71352    2.71827
4:    2.71781    2.71828
5:    2.71824    2.71828
6:    2.71828    2.71828
7:    2.71828    2.71828
```

Comment 1: We shall expand two functions, $u(x, y, z) = \exp(x)$ and $v(x, y, z) = \exp(x + y + z)$, both about the point $(x, y, z) = (0.5, 0.4, 0.0)$. The problem space is therefore three dimensional. We shall retain terms only up to degree seven.

Comment 2: In setting the points of evaluation, *the application program need not remember or explicitly refer to the reference point*: the DA variables know themselves where they were evaluated. (In fact, we even could have expanded u and v about two different reference points.)

Comment 3: In this loop we filter DA variables of various weights up to the maximum of seven. In this way we can follow the accuracy of the series as the number of terms increases. The reader should be able to explain easily the greater accuracy of one series over the other, as shown in the output.

3.3 Concatenation

The object of this exercise is to compute a derivative of two functions which have been concatenated together. The problem space is two dimensional, $\underline{w} = (x, y)$. Consider the two mappings,

$$\begin{aligned} \underline{a}(\underline{w}) &= \begin{pmatrix} xy^2 + \exp(x + y) \\ \dfrac{\cos(yx^2)}{x + 2} \end{pmatrix} , \\ \underline{b}(\underline{w}) &= \begin{pmatrix} \sin x \cos y \\ \dfrac{\exp(x^3)}{xy} \end{pmatrix} ; \end{aligned}$$

and their concatenation,

$$\underline{c}(\underline{w}) = \underline{b}(\underline{a}(\underline{w})) .$$

We shall calculate both components of $\partial^5 \underline{c} / \partial x^3 \partial y^2 |_{\underline{w}=(0,0)}$.

```
#include <std.h>
```

```
#include <stdio.h>
#include "da.hxx"

main() {
DA x, y, u, v, c1, c2;
DAVector a, b, c;
int index[2];
double answer[2];

DASetup( 2, 7, 2 );             // ** SEE COMMENT 1

x.setVariable(0);
y.setVariable(1);

u = x*y*y + exp( x + y );
v = cos( y*x*x ) / ( x + 2.0 );

printf( "Standard part of u is %lf\n", u.standardPart() );
printf( "Standard part of v is %lf\n", v.standardPart() );

a.setVariable( u, 0 );          // ** SEE COMMENT 2
a.setVariable( v, 1 );

DAFixReferenceAtEnd( a );       // ** SEE COMMENT 3
b.fixReference();               // ** SEE COMMENT 4

x.setVariable(0);               // ** SEE COMMENT 5
y.setVariable(1);

u = sin(x) * cos(y);            // ** SEE COMMENT 6
v = exp( x*x*x ) / ( x*y );

b.setVariable( u, 0 );
b.setVariable( v, 1 );

c = b*a;                        // ** SEE COMMENT 7

index[0] = 3;
index[1] = 2;
c.derivative( index, answer );

printf( "Zero: %lf\n", answer[0] );
printf( "One : %lf\n", answer[1] );
}
```

Output:
```
Standard part of u is 1.000000
Standard part of v is 0.500000
Zero: -25.215473
One : 50880.799265
```

Comment 1: For variety, I have here declared the DA variables *before* invoking **DASetup**. As long as it is invoked before using the variables, there is no problem (see Section 2.1).

Comment 2: This is the way that DAVectors get initialized. First their DA components are calculated and then inserted into the DAVector using the **.setVariable** method. The integer argument fixes the component associated with each prolonged number. After u and v are loaded into **a**, the connection between them is broken, and the program is free to use them again.

In fact, we do not need to use u and v at all; like **a**, b, and pb in Example 3.1 these intermediate variables only serve to separate logical steps in the program, making it easier to read. The statements,

```
    a.setVariable( x*y*y + exp( x + y ),        0 );
    a.setVariable( cos( y*x*x ) / ( x + 2.0 ),  1 );
```

would have worked just as well.

Comment 3: Anticipating concatenation, we change the default reference to that of the endpoint of **a**. That is, the default reference here gets set to $(1, 0.5)$, as indicated by the output of the **.standardPart**.

Comment 4: The reference point of b is set to the default reference, that is, to the endpoint of **a**. This is precisely what is needed to concatenate b with **a**.

Comment 5: The **.setVariable** method needs to be invoked by x and y again, because we have changed the reference point.

Comment 6: Notice that we simply write these functions as they have been defined. All information about reference points has been absorbed by the variables themselves; we do not need to use it explicitly in writing the rest of the program. The reference points of x and y were reset by the **.setVariable** method, and those of u and v are *automatically* adjusted by the = operator.

Comment 7: And here is the concatenation itself. All that remains is to access and print the derivative we need. You are invited to confirm at your leisure that the computed values are correct.

4 Status

A pre-compiled object library for MXYZPTLK, its source code, the necessary header file, da.hxx, and the LaTeXfile producing this document are all contained in a subdirectory of /home/dipole/michelotti on DIPOLE, part of the BOFFO-MYRTLE-FLIBBER-DIPOLE-QUAD domain of networked SUNs used by the Accelerator Division / Accelerator Physics group. These files are also readable from the VAX cluster ALMOND, but the current object library is for the SUN, not the VAX; by the time this document is released, a VAX object library should be available. Since Macintosh C++ exists, the source code should be portable without much difficulty (I hope) to the Mac II as well.

I have tested MXYZPTLK using a program that mimics the behavior of a (Hewlett-Packard) RPN calculator, but with registers containing DA variables rather than reals. These tests have been, of necessity, limited, and I would appreciate hearing from anyone interested in helping to extend them or who want to use the DA and DAVector classes in their own applications. Source code is also available to those who would like to help expand MXYZPTLK's capabilities; credit will be given in the **DAAbout** and **DANews** functions and in subsequent versions of this document for upgrades written by others.

Those who want to use MXYZPTLK can request the files through MICHELOTTI@FNALAD, ALMOND::MICHELOTTI, or (708) 840 4956. Please let me know of your experiences, especially if you have problems, discover bugs, upgrade the source code, or only have suggestions for improvements.

We shall end this note with a list of upgrades already under development.

- As already mentioned, the functions **pow** and **log** will be available in Version 1.1 of MXYZPTLK, which should be finished by the time this note is distributed.
- Links whose computed values are small enough to be considered zero to machine accuracy will be deallocated, as will those whose weights are higher than the maximum accurate weight after a differentiation.
- The requirement that all DA variables possess the same problem space will be relaxed. Using multiple calls to a new **DASetup** function, we should be able to use variables corresponding to different problem spaces, with different dimensions, in the same program. Of course, DA operations will automatically protect against unallowed mixtures.
- There is no need for DA variables to model prolongations of real-valued functions only. The concept can be extended naturally to functions of complex variables.
- A **.TeX** method will write a LaTeXexpression which is the multinomial representation of a DA variable in terms of user-specified symbols.
- When the DA variable is supposed to represent a symplectic mapping on the phase space of a Hamiltonian system, the symplectic condition can be enforced either at every stage of a computation or under the direct control of the application program using a **.symplectify** method.
- One important goal is to compile a library of user-friendly C++ functions, beginning with those that find periodic orbits of a Hamiltonian system and reduce it to normal form in their vicinities, as described in Reference [6]. It would be worthwhile to do this in the neighborhoods of resonant orbits as well, but finding those orbits is a more difficult task, requiring an interplay between mapping, analysis, and perhaps even graphics.[8]

ACKNOWLEDGEMENT

I am indebted to Herbert Wilf for a stimulating discussion of his ranking algorithm [14, 15, 9] which is at the core of evaluation and concatenation.

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] M. Berz. Differential algebra - a new tool. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

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

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

[5] Bruce Eckel. *Using C++* . Osborne McGraw-Hill, Berkeley, 1989.

[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), March 1989. To be published.

[7] Leo Michelotti. Differential algebras without differentials: an easy C++ implementation. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

[8] Leo Michelotti. Exploratory orbit analysis. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

[9] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. Academic Press, New York, 1978.

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

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

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

[13] Roy Thatcher. Programming in C - a word of caution. Fermilab Computing Division Newsletter, Vol. XVIII, No. 1, pp. 3-4, Jan-Feb 1990.

[14] Herbert S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.

[15] Herbert S. Wilf. A unified setting for selection algorithms (II). *Annals of Discrete Mathematics*, 2:135–148, 1978.

Fermi National Accelerator Laboratory

FERMILAB-FN-535
Revised

MXYZPTLK Version 3.1 User's Guide
A C++ Library for Differential Algebra

L. Michelotti

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

October 1995

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

MXYZPTLK Version 3.1 User's Guide: A C++ Library for Automatic Differentiation and Differential Algebra

Leo Michelotti

September, 1995

1 Introduction

If you need to calculate derivatives of complicated functions and find yourself either taking finite differences or writing the derivatives algebraically and then translating the expressions into source code, you may want to consider using automatic differentiation (AD). AD exploits the classic theorems of differential calculus to propagate information about derivatives through arithmetic operations. In this way, derivatives of a function can be calculated using the same program that calculates the function itself. Because no approximations are made, derivatives are calculated with machine accuracy, avoiding the errors inherent in finite differences, an especially important consideration when higher order derivatives are required.

MXYZPTLK is a library of C++ classes – or "objects" – for performing automatic differentiation. Originally written at Fermilab in 1989, with a "User's Guide" provided in 1990, it has undergone refinements and improvements over the last six years. It was originally announced outside Fermilab in *Automatic Differentiation of Algorithms: Theory, Implementation, and Application* (SIAM Press, 1991) and has been used in a variety of contexts. MXYZPTLK was the first implementation of AD which exploited object-oriented techniques (in C++) from the beginning.

Those who have not yet been exposed to AD/DA are invited to read the references in the bibliography. Here we will describe how to use Version 3.1 of MXYZPTLK, a C++ AD/DA library. In the next section we will explain quickly the mathematical models upon which the software is based. Section 3 contains a number of small programs demonstrating the use of AD/DA objects in MXYZPTLK. This document is motivated by the idea that people learn about objects more quickly by scanning a few examples of their use than by reading syntax rules governing their behavior: the reference section. Thus, Section 3 is its major piece, intended to jump start the reader. The shorter Section 4 will be the (still incomplete) "reference," devoted to describing the syntax for using objects, methods, and functions contained in the MXYZPTLK library.

2 Concepts

Let $f : R^N \to R$ and $g : R^N \to R$, be two "sufficiently" differentiable functions defined in an open neighborhood, $U \subset R^N$, of $\underline{u}_o \in U$. We will say that f and g are "n^{th} order equivalent," and write $f \approx_n g$, at $\underline{u}_o$ iff[1]

$$f(\underline{u}) = g(\underline{u}) + O(||\,\underline{u} - \underline{u}_o\,||^{n+1}) \quad .$$

This property is easily seen to be an equivalence relation among functions, which then enables us to define the equivalence class

$$\langle\, f, n, \underline{u}_o \,\rangle = \{\, g \mid f \approx_n g \quad \text{at } \underline{u}_o \,\} \quad . \tag{1}$$

which is called a "jet." It is identified by a triple containing a representative function, an integer, and a reference point.

The simplest element of any jet is a polynomial in the components of $(\underline{u} - \underline{u}_o)$. Let us define

$$\text{the operator} \quad D^{\underline{m}} = \prod_{k=0}^{N-1} \frac{1}{m_k!} \left(\frac{\partial}{\partial u_k} \right)^{m_k} \quad , \quad \text{and the shorthand} \quad \underline{a}^{\underline{m}} = \prod_{k=0}^{N-1} a_k^{m_k} \quad ,$$

where $\underline{m}$ is an array of N non-negative integers (the "index" array).[2] Let P be the polynomial satisfying,

$$P(\underline{u}) = \sum_{\underline{m}\,=\,\underline{0}}^{n} c_{\underline{m}} (\underline{u} - \underline{u}_o)^{\underline{m}}, \quad \text{where} \quad c_{\underline{m}} = (D^{\underline{m}} f)_{(\underline{u}_o)} \quad , \tag{2}$$

where the formal sum is taken over arrays of *non-negative* integers, $\underline{m}$ satisfying (a) $0 \le m_k \le n_k$, for all k, and (b) $\sum_k n_k = n$. With the usual assumptions about differentiability, it follows that $f \approx_n P$, and P can be used as the representative of the jet containing f. If this connection needs to be emphasized, we will write P_f for the polynomial.

We will interchangably refer to $\sum_k m_k$ as the degree of the polynomial term, its "weight," or the order of the associated derivative.

The important point is this: the equivalence property survives arithmetic operations. If $f_1 \approx_n f_2$ and $g_1 \approx_n g_2$ at $\underline{u}_o$, then $(\, f_1 \, op \, f_2 \,) \approx_n (\, g_1 \, op \, g_2 \,)$ at $\underline{u}_o$, where the operation symbol op stands for addition, subtraction, multipliciation, or division. Thus, to find the polynomial representative of the jet containing $f \, op \, g$ it suffices to perform the corresponding arithmetic operation on the polynomials equivalent to f and g and *truncate the answer at degree N.* This is called "truncated polynomial algebra," or "truncated power series algebra," and it is exactly what is needed to implement jet mathematics on a computer. We will refer to N as the "degree of truncation."

[1] This is an informal definition. It could easily be made more precise and incomprehensible. For example, something like: $\exists C \in R \, \exists U \in R^N \, \forall \underline{u} \in U : |f(\underline{u}) - g(\underline{u})| < C||\underline{u} - \underline{u}_o||^{n+1}$. However, there is no excuse here for this level of formality.

[2] It is a nuisance to start the product at "$k = 0$." This is done to maintain consistence with the C and C++ array convention.

Addition and subtraction are the operations easiest to implement. We merely add the corresponding coefficients of the truncated polynomials.

$$D^{\underline{m}}(f \pm g) = D^{\underline{m}}f \pm D^{\underline{m}}g \quad . \tag{3}$$

Multiplication is accomplished easily using Leibniz's rule,

$$D^{\underline{n}}(f \cdot g) = \sum_{\underline{m}=\underline{0}}^{\underline{n}} (D^{\underline{m}}f)\,(D^{\underline{n}-\underline{m}}g) \quad , \tag{4}$$

Truncation means that $\sum_k n_k <= N$. Division is accomplished by a form of repeated multiplication. Notice that by combining $f \cdot g = w$ with Eq.(4) we can write a recursive procedure for defining the higher orders of $\hat{v}$ in terms of its lower orders,

$$D^{\underline{n}}g = \frac{1}{f}\left[D^{\underline{n}}w - \sum_{\underline{m}=\underline{0}}^{\underline{n}} (D^{\underline{m}}f)\,(D^{\underline{n}-\underline{m}}g) \right] \tag{5}$$

starting with $g = w/f$. Eqs.(3), (4), and (5) form the basis for MXYZPTLK's arithmetic algorithms.

Building on jets, MXYZPTLK includes an object for modeling the action of Lie operators. For this, the "problem space" of coordinates is partitioned into two subspaces, as determined by the application: a "phase space," or "dynamical sector," of dimension N_d, whose coordinates we will write as $\underline{u}$, and a "control sector," of dimension $N_c = N - N_d$, with coordinates written as $\underline{a}$. For example, suppose that we are studying the restricted three-body problem: say, the motion of a small satellite under the influence of a planet and its moon. The dynamical sector would represent the six-dimensional phase space, corresponding to the initial momentum and position of the satellite. However, if we wanted to examine such questions as the sensitivity of the final state to the masses of the planet and moon, then we would add these as "control" coordinates of the problem space. The "index array" associated with a derivative is the ordered array of integers which specify the derivative. For example, if the problem space is 3 dimensional, say $\underline{x} = (u_0, u_1, u_2)$, then the index array associated with $\partial^6 f(\underline{u})/\partial u_0 \partial u_1^3 \partial u_2^2$ would be $(1, 3, 2)$. The sum of the indices (the components of an index array) will be varyingly referred to as the derivative's "weight," its "order," or its "degree."

By design, Lie operators act only on the dynamical coordinates. In the context of this discussion, a mathematical Lie operator can be defined as a differential operator of the form,

$$\mathbf{V} = \underline{v}(\underline{u}, \underline{a}) \cdot \frac{\partial}{\partial \underline{u}} \quad . \tag{6}$$

Of particular importance is the exponential map, which maps functions onto functions, formally obtained by the expression,

$$g = e^{\mathbf{V}} f = \sum_{k=1}^{\infty} \frac{1}{n!} \mathbf{V}^n f \quad . \tag{7}$$

Notice that if $f \approx_n g$, then in general, $\mathbf{V}f \approx_{n-1} \mathbf{V}g$. Intepreted as acting on a jet, a Lie operator will lower its order. This can be mitigated restricting consideration to Lie operators whose $\underline{v}$, defined in Eq.(6), satisfies

$$\underline{v}(\underline{u}, \underline{a}) = O(\,||\underline{u} - \underline{u}_o||\,) \quad .$$

This is essential in order to implement an exponential map, with its repeated application of $\mathbf{V}$. In addition, for an exact implementation, we should require that

$$\underline{v}(\underline{u}, \underline{a}) = O(\,||\underline{u} - \underline{u}_o||^2\,) \quad .$$

This condition means that the nonvanishing term of lowest degree in P_f has smaller degree than the corresponding term in $P_{\mathbf{V}f}$. Upon repeated application of $\mathbf{V}$, the lowest nonvanishing degree eventually becomes larger than the order of the jet, and all but a finite number of terms in Eq.(7) can be ignored. This condition provides Lie operators that convert AD into an exact differential algebra (DA). We will illustrate in the next section how these operators are implemented in MXYZPTLK.

A mapping can be thought of as an array of N_d functions,

$$\underline{f} : R^{N_d} \times R^{N_c} \rightarrow R^{N_d} \quad .$$

Alternatively, we can write this as an array of functions,

$$\underline{f} : R^N \times R^N \quad ,$$

which acts as the identity on the control sector. This approach is formally more convenient when one wants to consider concatenating mappings: $\underline{h} = \underline{g} \circ \underline{f}$. Written with arguments, $\underline{h}(\underline{u}, \underline{a}) = \underline{g}(\underline{f}(\underline{u}, \underline{a}), \underline{a})$ in the first picture becomes the more natural $\underline{h}(\underline{z}) = \underline{g}(\underline{f}(\underline{z}))$ in the second, where $\underline{z} = (\underline{u}, \underline{a})^T$.

Because we are going to model an algebra of functions, of special importance are the coordinate functions themselves, which are projections onto the components of $\underline{u}$. For example, if $\underline{u} = (u_0, u_1, u_2)^T$, we could define the coordinate functions x, y, and z according to $x(\underline{u}) = u_0$, $y(\underline{u}) = u_1$, and $z(\underline{u}) = u_2$. We could then write a new function, say

$$f = e^{-x^2} \sin y \quad ,$$

and this is interpreted as an equation relating functions to functions. Notice that it would be *incorrect* to write,

$$f(x, y) = e^{-x^2} \sin y \quad ?? \quad ,$$

as this would have a completely different meaning, in fact, no meaning at all in the current context. Instead, we can write something like,

$$f(\underline{u}) = \left(e^{-x^2} \sin y \right)_{(\underline{u})} , \quad \text{for all } \underline{u} \quad .$$

Functions are evaluated numerically, not symbolically, as jets. The triplet shown in Eq.(1) is stored, with f given by the truncated polynomial representation of Eq.(2). For the example given above, we would begin the calculation with the coefficients,

$$\begin{aligned} x(\underline{u}) &\rightarrow c_{(0,0,0)} = 0, c_{(1,0,0)} = 1 \\ y(\underline{u}) &\rightarrow c_{(0,0,0)} = 0, c_{(0,1,0)} = 1 \\ z(\underline{u}) &\rightarrow c_{(0,0,0)} = 0, c_{(0,0,1)} = 1 \quad . \end{aligned}$$

Numerical jets are built from such starting points using the rules of Eqs.(3), (4), and (5). Of course, this all happens internally and is transparent to the user, who simply writes an application as though using ordinary double precision variables. Programs implementing this example and others are provided in the next section for illustration.

One final note: there is no reason to restrict consideration to real coordinates. What we have written for real functions can be extended to complex functions as well. Such an extension was indeed included in MXYZPTLK for the purpose of doing normal form calculations conveniently.

3 Examples

MXYZPTLK contains the classes `Jet`, `coord`, `LieOperator`, `Map`, and their complex counterparts `JetC`, `coordC`, `CLieOperator`, and `CMap`. We display below a few sample programs which illustrate various features of the MXYZPTLK library. It is hoped that they are sufficiently instructive to act as prototypes for your own calculations.

No examples were included involving arithmetic on `Map` and `LieOperator` objects, but it can be done notwithstanding. They possess the properties of a vector space. It is possible to add and subtract `Map`s and `LieOperator`s together, and to multiply them by `double`, `complex`, or `Jet` objects.

3.1 Evaluating a derivative

This first demo simply prints the value of the derivative,

$$\partial(e^{-x^2}\sin y)/\partial x^m\,\partial y^n|_{x_o,y_o} \quad ,$$

where the parameters x_o, m, y_o, n are entered on the command line. The source code is shown below, followed by a few sample uses and commentary.

Source: dfr.cc

```
#include "mxyzptlk.rsc"

main( int argc, char** argv ) {

 if( argc != 5 ) {
  cout << "\nUsage: " << argv[0]
       << "  x n_x y n_y\n"
       << endl;
  exit(0);
 }

 int deg [2];
 deg[0] = atoi( argv[2] );
 deg[1] = atoi( argv[4] );

 Jet::Setup( 2, deg[0] + deg[1] );

 coord x( atof( argv[1] ) ), y( atof( argv[3] ) );

 cout << "Answer: "
      << ( exp(-x*x) * sin(y) ).derivative( deg )
16        << endl;

17  }
```

Output

```
hazel 1: dfr

Usage: dfr  x n_x y n_y

hazel 2: dfr 0 4 0 5
Answer: 12
hazel 3: dfr 1 3 -1 7
Answer: -0.795064
```

Comments

Line 1: The header file mxyzptlk.rsc must be included near the top of any MXYZPTLK user program.

Lines 3-8: Prints a little "usage" message if the program name is written without arguments. (See the "hazel 1" prompt above.)

Lines 9-11: The integer array `deg` will carry the indices of the desired derivative; that is, it carries m and n. *The ordering is determined by the order in which the* `coord` *variables have been declared.* In this case, x came first, so `x` is internally associated with index 0, and `y`, with index 1. Thus, to find $\partial(e^{-x^2} \sin y)/\partial x^m \partial y^n |_{x_o,y_o}$, we set `deg[0]` to m and `deg[1]` to n before using it as the argument to the **.derivative** member function, in Line 15.

Line 12: The routine **Jet::Setup** must be called before performing AD/DA manipulations. In this call, the first argument tells the library the number of independent variables, and the second indicates the maximum order of derivative desired. Since the indices, m and n, have been given on the command line and entered into the array, `deg`, the second argument is set to their sum.

Line 13: Variables `x` and `y` are declared as **coord** objects, or "coordinates." **coord**s are the most basic building blocks for AD calculations, the "independent variables" of the function to be differentiated. They implement the projection functions described at the end of Section 3. The two arguments from the command line set their values, which in turn determine the point at which the function to be constructed will be differentiated.

Lines 14-16: Finally, the function $e^{-x^2} \sin y$ is constructed, and, in the same line, the requested derivative is sent to the output stream. Notice that arithmetic operations on bf coord objects do not return bf coord objects; they return **Jet** objects. (A **coord** is, of course, just a special kind of **Jet**.)

3.2 Jets

The previous example was simple enough that there was no need to store the calculation in a variable. If it is necessary or desireable to do so, the appropriate variable type is called a `Jet`. This example shows Jets being used both to store the results of calculations and to return them from functions.

Source: g5.cc

```
#include "mxyzptlk.rsc"

Jet g( const Jet& x, int n ) {
 Jet z = 0.0;
 Jet term;
 term = x;
 for( int k = 1; k <= n; k++ ) {
  z += term / ( (double) k );
  term *= x;
 }
 z.stacked = 1;
 return z;
}

main() {
 Jet::Setup( 3, 6 );

 coord x(0.0), y(0.0), z(0.0);

 Jet a;
 a = x*y + y*z + z*x;
 a.printCoeffs();
 ( g( a, 3 )*g( sin(a), 5 ) ).printCoeffs();

}
```

Output:

```
hazel 1: g5

Count  = 4, Weight = 2, Max accurate weight = 6
Reference point:
0.000000e+00  0.000000e+00  0.000000e+00

Index:    0  0  0   Value:   0.000000e+00
Index:    0  1  1   Value:   1.000000e+00
Index:    1  0  1   Value:   1.000000e+00
```

```
Index:    1  1  0   Value:   1.000000e+00



Count  = 17, Weight = 6, Max accurate weight = 6
Reference point:
0.000000e+00  0.000000e+00  0.000000e+00

Index:    0  0  0   Value:   0.000000e+00
Index:    0  2  2   Value:   1.000000e+00
Index:    1  1  2   Value:   2.000000e+00
Index:    1  2  1   Value:   2.000000e+00
Index:    2  0  2   Value:   1.000000e+00
Index:    2  1  1   Value:   2.000000e+00
Index:    2  2  0   Value:   1.000000e+00
Index:    0  3  3   Value:   1.000000e+00
Index:    1  2  3   Value:   3.000000e+00
Index:    1  3  2   Value:   3.000000e+00
Index:    2  1  3   Value:   3.000000e+00
Index:    2  2  2   Value:   6.000000e+00
Index:    2  3  1   Value:   3.000000e+00
Index:    3  0  3   Value:   1.000000e+00
Index:    3  1  2   Value:   3.000000e+00
Index:    3  2  1   Value:   3.000000e+00
Index:    3  3  0   Value:   1.000000e+00
```

Comments:

Lines 17-18: Here we declare the variable `a` to be of type Jet and set its value to be the symmetric polynomial $xy + yz + zx$.

Line 19: The member function **Jet::printCoeffs()** prints the coefficients of the truncated polynomial. This command, "a.printCoeffs()," results in the first seven (non-void) lines of output. "Count = 4" means that there are four terms retained in the polynomial. "Weight = 2" tells us that the degree of the polynomial is 2, while "Max accurate weight = 6" indicates that **Jet::Setup** requested terms of highest degree 6 were to be carried. The Jet's reference point, $(0, 0, 0)$, shown in the next two lines of output, was set in Line 16 of the source, when the coord variables, x, y, and z were declared. Finally, the list of indices and values record the terms of the polynomial: indices represent the exponents and values, the coefficients. Thus, because of the ordering, the line "Index: 0 1 1 Value: 1.000000e+00" tells us that the $x^0y^1z^1$ term of `a` has coefficient 1. The next two lines provide the same information for the $x^1y^0z^1$ and $x^1y^1z^0$ terms. In other words, `a` models the polynomial, $xy + xz + yz$, as it should.

Lines 2-12: These lines define a Jet function that computes the polynomials,

$$g_n(x) = \sum_{k=1}^{n} x^k/k$$

We put off a discussion of the cryptic Line 10 until later.

Line 20: Here we print the information about the polynomial,

$$g_3(a)g_5(\sin a), \quad \text{where} \quad a = xy + yz + zx \quad ,$$

truncated at degree 6 (see Source Line 14). The Jet function `g` is invoked twice, the results multiplied together, and the member function **Jet::printCoeffs()** invoked, which, as before, sends the result to the output stream. This results in the second chunk of output, which indicates the polynomial, $y^2z^2 + 2xyz^2 + 2xy^2z + x^2z^2 + 2x^2yz + x^2y^2 + y^3z^3 + 3xy^2z^3 + 3xy^3z^2 + 3x^2yz^3 + 6x^2y^2z^2 + 3x^2y^3z + x^3z^3 + 3x^3yz^2 + 3x^3y^2z + x^3y^3$. I leave it to the reader to explain why all the coefficients are integers and to determine whether this remarkable property extends to terms of higher degree.

Lines 4-5 and 17-18: The Jets `term` and `a` are first declared and then assigned values, in separate lines. Each could have been combined into one line, as in "Jet term = x; ."[3] However, there is a subtle reason for not doing this. Compilers interpret the statement "Jet term = x;" as equivalent to "Jet term(x);" and invoke the copy constructor, *not* the member function **Jet::operator=**, to assign the value. Because of the way Jet variables store data,[4] this may result in an error. Therefore, *it is recommended that Jets always be declared and assigned values on separate lines.*

3.3 Differentiation

In Section 3.1 we printed the value of a particular derivative of a function. The corresponding AD *operation* is to take the derivative of a function, thereby creating a new function. The Jet method which does this is **Jet::D**. We will illustrate its use by calculating coefficients of the Hermite polynomials,

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2} \quad . \tag{8}$$

Source: Hermite.cc

```
 1  #include "mxyzptlk.rsc"

 2  main( int argc, char** argv ) {

 3   if( argc != 2 ) {
 4    cout << "\nUsage: " << argv[0] << "  n"
 5         << endl;
 6    exit(0);
 7   }
```

[3] I defy you to punctuate this sentence correctly.
[4] The envelope-letter idiom is employed.

```
  int n = atoi( argv[1] );

  Jet::Setup( 1, 2*n );
  coord x( 0.0 );
  Jet f, g;
  int d = 1;

  f = exp( - x*x );
  g = f;

  int k = 0;
  cout << "Results for k = " << k << endl;
  ( g / f ).printCoeffs();
  for( k = 1; k <= n; k++ ) {
    g = - g.D( &d );
    cout << "Results for k = " << k << endl;
    ( g / f ).printCoeffs();
  }

}
```

Output:

```
hazel 1: Hermite 4
Results for k = 0

Count  = 1, Weight = 0, Max accurate weight = 8
Reference point:
0.000000e+00

Index:    0   Value:   1.000000e+00


Results for k = 1

Count  = 2, Weight = 1, Max accurate weight = 7
Reference point:
0.000000e+00

Index:    0   Value:   0.000000e+00
Index:    1   Value:   2.000000e+00


Results for k = 2
```

```
Count  = 3, Weight = 8, Max accurate weight = 6
Reference point:
0.000000e+00

Index:    0   Value:  -2.000000e+00
Index:    2   Value:   4.000000e+00


Results for k = 3

Count  = 4, Weight = 7, Max accurate weight = 5
Reference point:
0.000000e+00

Index:    0   Value:   0.000000e+00
Index:    1   Value:  -1.200000e+01
Index:    3   Value:   8.000000e+00


Results for k = 4

Count  = 5, Weight = 8, Max accurate weight = 4
Reference point:
0.000000e+00

Index:    0   Value:   1.200000e+01
Index:    2   Value:  -4.800000e+01
Index:    4   Value:   1.600000e+01
```

Comments:

Line 9: The value of n has been entered on the command line. The reason for setting the second argument of **Jet::Setup** to $2n$ rather than n will be explained shortly.

Line 10: It would be a useful exercise to understand why the reference point must be zero. How would the results change if a different reference were chosen? (Try it!)

Lines 13-14: `f` is assigned the jet containing e^{-x^2}. `g` will take on the values $(-1)^k \frac{d^k}{dx^k} e^{-x^2}$, for $k = 0 \ldots n$, so we begin by setting it equal to `f`.

Line 19: Here is where the differentiation is done. In each step of the loop (Lines 18-22), a single derivative of `g` is taken and stored back into `g`. The order of the derivative is determined by the argument of **Jet::D**, which is an array of doubles, just like that of **Jet::derivative**. In this case, since the problem space is one dimensional, the address, `&d`, serves the same purpose as the name of an array. Notice that the same effect, apart from the sign, would have been obtained somewhat less efficiently by substituting "`g = f.D(&k)`" for Line 19.

Lines 17 and 21: These are the output lines. Looking back to Eq.(8), we see that it is only necessary to print out the coefficients from `g/f` to obtain the desired polynomials. Looking at the output, we identify the polynomials.

$$
\begin{aligned}
H_0(x) &= 1 \\
H_1(x) &= 2x \\
H_2(x) &= 4x^2 - 2 \\
H_3(x) &= 8x^3 - 12x \\
H_4(x) &= 16x^4 - 48x^2 + 12
\end{aligned}
$$

Line 9 (again): Now let us return to the arguments of **Jet::Setup**. A **Jet** variable carries polynomial coefficients only up to the particular order determined by **Jet::Setup**. When a derivative operation is performed, the degree of the representative polynomial decreases by one. This is reflected in the "Max accurate weight" field in the output. We begin with a jet of degree 8. At each step through the loop, a single differentiation is done, so that by the end we have a jet of "maximum accurate weight" 4, corresponding to the degree of the requested polynomial. The information about accuracy is carried through arithmetic operations, so that the "maximum accurate weight" of `g/f` is automatically determined by `g`, not `f`. *Had we begun with a jet of smaller degree, the final polynomial would not have contained all the coefficients required.* Thus the argument of **Jet::Setup** was determined by our prior knowledge that $H_n(x)$ is a polynomial of degree n.

3.4 Maps and Jacobians

A **Map** is an object that models a multi-dimensional differentable function: $\phi : R^n \rightarrow R^n$. This example prints the Jacobian matrix of the transformation from Cartesian to polar coordinates, $\partial(x, y, z)/(r, \theta, \phi)$, and its inverse, $\partial(r, \theta, \phi)/(x, y, z)$, at a point specified on the command line.

Source: survey.cc

```
 1  #include "mxyzptlk.rsc"

 2  main( int argc, char** argv ) {

 3   if( argc != 4 ) {
 4    cout << "\nUsage: " << argv[0]
 5         << "  <r>  <theta (deg)>  <phi (deg)>\n"
 6         << endl;
 7    exit(0);
 8   }

 9   const double d2r = M_PI / 180.0;
```

```
10   MatrixD M( 3, 3 );
11   Jet::Setup( 3, 1 );

12   coord r     (     atof( argv[1] ) ),
13         theta ( d2r*atof( argv[2] ) ),
14         phi   ( d2r*atof( argv[3] ) );
15   Map   position;

16   position.SetComponent( 0, r * sin( theta ) * cos( phi ) );
17   position.SetComponent( 1, r * sin( theta ) * sin( phi ) );
18   position.SetComponent( 2, r * cos( theta )              );

19   M = position.Jacobian();
20   cout << M << "\n\n" << M.inverse() << endl;
21  }
```

Output:

```
hazel 1: survey 1. 30. 45.
( 0.35355339, 0.61237244, -0.35355339,  )
( 0.35355339, 0.61237244,  0.35355339,  )
( 0.8660254, -0.5,         0,           )

( 0.35355339, 0.35355339,  0.8660254,   )
( 0.61237244, 0.61237244, -0.5,         )
( -1.4142136, 1.4142136,   0,           )
```

Comments:

Lines 12-14: The coordinates (r, θ, ϕ) are read from the command line and **coord** variables are declared with these values. Multiplication by `d2r` merely converts from degrees to radians.

Lines 15-18: A **Map** variable is declared and its components set. `position` is to model the function,

$$\phi : (r, \theta, \phi) \mapsto (x, y, z) = (r \sin\theta \cos\phi, r \sin\theta \sin\phi, r \cos\theta) \quad .$$

The member function **Map::SetComponent** is used to set the corresponding components of `position`.

Lines 10, 19-20 A 3×3 **Matrix** of double precision numbers, `M`, is declared, and the member function **Map::Jacobian()** is used to load `M` with the Jacobian of `position`. Finally, in Line 20, the Matrix and its inverse are sent to the output stream.

3.5 Evaluation

A `Jet` variable models a mathematical Jet by containing the coefficients of its polynomial representative. The member function `Jet::operator()` provides a mechanism for evaluating that polynomial. We'll illustrate that by evaluating $e^{-x^2} \sin y$ using its Jet representative and comparing to the exact value.

Source: ev.cc

```
#include "mxyzptlk.rsc"

main( int argc, char** argv ) {

 if( argc != 4 ) {
  cout << "\nUsage: " << argv[0]
       << "  x y n\n"
       << endl;
  exit(0);
 }

 int deg = atoi( argv[3] );
 Jet::Setup( 2, deg );

 coord x( atof( argv[1] ) ), y( atof( argv[2] ) );
 Jet f = exp(-x*x) * sin(y);

 double point[2];
 while(1) {
   cout << "Enter x and y: ";
   cin  >> point[0] >> point[1];
   cout << "Jet answer: "
        << f( point )
        << "  Exact answer: "
        << sin( point[1] ) * exp( - point[0]*point[0] )
        << endl;
 }

}
```

Output:

```
hazel 1: ev 1 1 16
Enter x and y: 1 1
Jet answer: 0.30956  Exact answer: 0.30956
Enter x and y: 0.4 1.6
Jet answer: 0.85178  Exact answer: 0.85178
```

```
Enter x and y: 2 2
Jet answer: 0.0166206  Exact answer: 0.0166544
Enter x and y: 0 0
Jet answer: 4.05366e-05  Exact answer: 0
Enter x and y: ^C

hazel 2: ev -0.5 1.5 8
Enter x and y: -1.0 1.0
Jet answer: 0.309656  Exact answer: 0.30956
Enter x and y: 0 0
Jet answer: -0.00358171  Exact answer: 0
Enter x and y: -1.5 0.5
Jet answer: 0.0870872  Exact answer: 0.0505311
Enter x and y: ^C
```

Comments:

Line 11: As in Section 3.1, the reference point is specified on the command line of the program. `coord` variables are set in preparation for calculations.

Lines 12 and 18: Most of this program is similar to what we've seen already. Line 18 contains the new operation. After `f` is constructed in Line 12, it is used in Line 18 to evaluate the polynomial that it represents. The loop in Lines 14-22 repeats indefinitely, and the results for several values of `point` can be seen in the **Output**. On the command line, we have specified that $(x, y) = (1, 1)$ be the reference point of the problem, and that the representative polynomials be truncated at degree 16. *The rest of the program need not use the reference point explicitly.* In particular, Line 18 makes no mention of it. Thus, for example, to find $f((0.4, 1.6))$ we enter 0.4 and 1.6, as would be most natural. The Jet itself knows its own reference point and subtracts it automatically before evaluating the polynomial.

3.6 Filters

Filters are available to create new Jets by selecting a subset of the coefficients contained in an already existing Jet. The most basic filter simply selects coefficients whose weights lie with a given range. We illustrate that by calculating e using two different power series.

Source: evaltest.cc

```
 1  #include "mxyzptlk.rsc"

 2  main() {
 3   double r[3], s[3];

 4   Jet::Setup( 3, 7 );
```

```
coord x(0.5), y(0.4), z(0.0);
Jet u, v;

u = exp( x );
v = exp( x + y + z );

r[0] = 1.0;   s[0] = 0.33;
r[1] = 0.0;   s[1] = 0.33;
r[2] = 0.0;   s[2] = 1.0 - s[0] - s[1];

for( int w = 1; w <= 7; w++ ) {
  printf( "%d: %lf   %lf  \n",
          w,
          (u.filter( 0, w ))( r ),
          (v.filter( 0, w ))( s )
        );
  }
}
```

Output

```
hazel 1: evaltest
1: 2.473082   2.705563
2: 2.679172   2.717861
3: 2.713520   2.718271
4: 2.717814   2.718282
5: 2.718243   2.718282
6: 2.718279   2.718282
7: 2.718282   2.718282
```

Comments

Comment 1: We shall expand two functions, $u(x, y, z) = \exp(x)$ and $v(x, y, z) = \exp(x + y + z)$, both about the point $(x, y, z) = (0.5, 0.4, 0.0)$. The problem space is therefore three dimensional. We shall retain terms only up to degree seven.

Comment 2: In setting the points of evaluation, *the application program need not remember or explicitly refer to the reference point*: the Jet variables know themselves where they were evaluated. (In fact, we even could have expanded u and v about two different reference points.)

Comment 3: In this loop we filter Jet variables of various weights up to the maximum of seven. In this way we can follow the accuracy of the series as the number of terms increases. The reader should be able to explain easily the greater accuracy of one series over the other, as shown in the output.

3.7 Concatenation

The object of this exercise is to compute a derivative of two functions which have been concatenated together. The problem space is two dimensional, $\underline{u} = (x, y)^T$. Consider the two mappings,

$$\underline{a}(\underline{u}) = \begin{pmatrix} xy^2 + \exp(x+y) \\ \dfrac{\cos(yx^2)}{x+2} \end{pmatrix} , \tag{9}$$

$$\underline{b}(\underline{u}) = \begin{pmatrix} \sin x \cos y \\ \dfrac{\exp(x^3)}{xy} \end{pmatrix} , \tag{10}$$

and their composition,

$$\underline{c}(\underline{u}) = \underline{b}(\underline{a}(\underline{u})) . \tag{11}$$

We shall calculate both components of $\partial^5 \underline{c} / \partial x^3 \partial y^2 |_{\underline{w}=(0,0)}$.

Source: concattest.cc

```
#include "mxyzptlk.rsc"

main() {
  Jet         q, v, w, z;
  static int  index[] = { 3, 2 };
  double      answer[2];

  Jet::Setup( 2, 7, 2 );

  coord x(0.0), y(0.0);
  Map   a, b;

  a.SetComponent( 0, q = x*y*y + exp( x + y )       );
  a.SetComponent( 1, v = cos( y*x*x ) / ( x + 2.0 ) );

  w = sin(q) * cos(v);
  z = exp( q*q*q ) / ( q*v );

  x.set( a(0).standardPart() );
  y.set( a(1).standardPart() );
  b.fixReferenceAtEnd( a );

  b.SetComponent( 0, sin(x) * cos(y)       );
  b.SetComponent( 1, exp( x*x*x ) / ( x*y ) );

  b(a) .derivative( index, answer );

  cout << "Using composition:       " << answer[0]              << "  "
20                                      << answer[1]            << endl;
21    cout << "Using explicit formulas: " << w.derivative( index )  << "  "
22                                      << z.derivative( index )  << endl;
23  }
```

Output:

```
hazel 1: concattest
Using composition:       -25.2155  50880.8
Using explicit formulas: -25.2155  50880.8
```

Comments:

Line 7: There was really no need to carry terms of degree seven for this calculation: five would have been sufficient. Notice that **Jet::Setup** is invoked *after* `Jet` and `Map` variables have been declared in Lines 3-4. This is not good practice, but it was done here to illustrate this particular capability of MXYZPTLK. It is permitted to declare variables before invoking **Jet::Setup**. This allows for the possibility of giving `Jet` variables global scope. It is only necessary that **Jet::Setup** be used before carrying out operations on these variables.

Lines 9-10 and 16-17: The `Jets` `a` and `b` are initialized so as to model the mappings $\underline{a}$ and $\underline{b}$ appearing in Eqs.(9) and 10. In the process of doing that, the components of `a` are loaded into the `Jets` `q` and `v` for later use.

Lines 13-15: Before setting the components of `b`, the reference point must be adjusted. When `a` was declared, it was assigned the same reference point as `a`, a reference determined by the declarations in Line 7. However, since we are going to concatenate `b` with `a` to form `b(a)`, if $(0,0)$ is the reference point of `a`, then we must reset the reference point of `b` to be `a`$((0,0))$. This is done with the member functions **Map::fixReferenceAtEnd**, which adjusts the reference point of a mapping to the image of the reference point of another mapping. In addition, the coordinates `x` and `y` are reset to the new reference point, using **coord::set**, prior to their reuse Member function **Jet::standardPart** returns the polynomial coefficient $c_{\underline{0}}$ and is used here to find `a`$((0,0))$.

Line 18: First the two `Jets` `b` and `a` are composed, in accordance with Eq.(11). The member functions **Map::derivative** is used to load the desired derivatives into the array `answer`.

Lines 19-23: Using `b(a)` and using `w` and `z` are compared. The results should be, and are, identical. Notice the differences between **Jet::derivative** and **Map::derivative**. The latter evaluates the desired derivative for each component of the `Map` and returns the resulting numbers in an array argument.

3.8 Inversion

If the function $f : R^n \rightarrow R^n$ is invertible at the reference point $\underline{u}_o$, then the member function **Map::Inverse** allows one to compute the (multidimensional) Jet corresponding to the local inverse map, f^{-1} at the reference point $f(\underline{u}_o)$. In the demo below we invert the function

$$f : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} 3 + x + 3y + xy - yz \\ -1 + y - x + z + xz + y^2 \\ 2 + z + 2x + yz - xyz \end{pmatrix}$$

at $\underline{u}_o = (-1.2, 3.5, 2.1)^T$. The **Output** section is rather lengthy, but comments do indeed follow it, as usual.

Source:

```
#include "mxyzptlk.rsc"

main() {

 Jet::Setup( 3, 4, 3 );

 coord  x(-1.2), y(3.5), z(2.1);
 Map    f, u;

 f.SetComponent( 0,  3.0 + x + 3.0*y + x*y - y*z   );
 f.SetComponent( 1, -1.0 + y - x + z + x*z + y*y   );
 f.SetComponent( 2,  2.0 + z + 2.0*x + y*z - x*y*z );

 u = f.Inverse();

 cout << "\n====== f.printCoeffs(); ===================\n" << endl;
 f.printCoeffs();
 cout << "\n====== u.printCoeffs(); ===================\n" << endl;
 u.printCoeffs();
 cout << "\n====== f(u).printCoeffs(); ================\n" << endl;
 f(u).printCoeffs();
 cout << "\n====== u(f).printCoeffs(); ================\n" << endl;
 u(f).printCoeffs();
}
```

Output:

```
====== f.printCoeffs(); ===================

************ Begin LieOperator::printCoeffs ********
Weight: 3

******************
**** Component index = 0
```

```
Count  = 6, Weight = 2, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:   7.500000e-01
Index:    0  0  1   Value:  -3.500000e+00
Index:    0  1  0   Value:  -3.000000e-01
Index:    1  0  0   Value:   4.500000e+00
Index:    0  1  1   Value:  -1.000000e+00
Index:    1  1  0   Value:   1.000000e+00



******************
**** Component index = 1

Count  = 6, Weight = 2, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:   1.553000e+01
Index:    0  0  1   Value:  -2.000000e-01
Index:    0  1  0   Value:   8.000000e+00
Index:    1  0  0   Value:   1.100000e+00
Index:    0  2  0   Value:   1.000000e+00
Index:    1  0  1   Value:   1.000000e+00



******************
**** Component index = 2

Count  = 8, Weight = 3, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:   1.787000e+01
Index:    0  0  1   Value:   8.700000e+00
Index:    0  1  0   Value:   4.620000e+00
Index:    1  0  0   Value:  -5.350000e+00
Index:    0  1  1   Value:   2.200000e+00
Index:    1  0  1   Value:  -3.500000e+00
Index:    1  1  0   Value:  -2.100000e+00
Index:    1  1  1   Value:  -1.000000e+00
```

```
************ End   LieOperator::printCoeffs ********

====== u.printCoeffs(); ===================


************ Begin LieOperator::printCoeffs ********
Weight: 4

******************
**** Component index = 0

Count  = 35, Weight = 4, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01

Index:    0  0  0   Value:  -1.200000e+00
Index:    0  0  1   Value:   1.842162e-01
...
Index:    3  0  1   Value:   1.291290e-01
Index:    3  1  0   Value:  -6.330736e-02
Index:    4  0  0   Value:   5.789563e-02



******************
**** Component index = 1

Count  = 35, Weight = 4, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01

Index:    0  0  0   Value:   3.500000e+00
Index:    0  0  1   Value:  -1.936699e-02
...
Index:    3  1  0   Value:   1.901941e-02
Index:    4  0  0   Value:  -1.652798e-02



******************
**** Component index = 2

Count  = 35, Weight = 4, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01
```

```
Index:    0  0  0   Value:   2.100000e+00
Index:    0  0  1   Value:   2.385095e-01
...
Index:    3  1  0   Value:  -8.390785e-02
Index:    4  0  0   Value:   7.560193e-02



************ End   LieOperator::printCoeffs ********

====== f(u).printCoeffs(); ================


************ Begin LieOperator::printCoeffs ********
Weight: 1

******************
**** Component index = 0

Count  = 2, Weight = 1, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01

Index:    0  0  0   Value:   7.500000e-01
Index:    1  0  0   Value:   1.000000e+00



******************
**** Component index = 1

Count  = 2, Weight = 1, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01

Index:    0  0  0   Value:   1.553000e+01
Index:    0  1  0   Value:   1.000000e+00



******************
**** Component index = 2

Count  = 2, Weight = 1, Max accurate weight = 4
Reference point:
7.500000e-01  1.553000e+01  1.787000e+01
```

```
Index:    0  0  0   Value:   1.787000e+01
Index:    0  0  1   Value:   1.000000e+00



************ End   LieOperator::printCoeffs ********

====== u(f).printCoeffs(); ================


************ Begin LieOperator::printCoeffs ********
Weight: 4

******************
**** Component index = 0

Count  = 3, Weight = 4, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:  -1.200000e+00
Index:    1  0  0   Value:   1.000000e+00
Index:    0  4  0   Value:  -4.786373e-15



******************
**** Component index = 1

Count  = 3, Weight = 4, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:   3.500000e+00
Index:    0  1  0   Value:   1.000000e+00
Index:    0  4  0   Value:   9.367778e-16



******************
**** Component index = 2

Count  = 3, Weight = 4, Max accurate weight = 4
Reference point:
-1.200000e+00  3.500000e+00  2.100000e+00

Index:    0  0  0   Value:   2.100000e+00
```

```
Index:    0  0  1   Value:   1.000000e+00
Index:    0  4  0   Value:  -3.265400e-15

************ End   LieOperator::printCoeffs ********
```

Comments:

Lines 5-10: The `Map` variable `f` is constructed using methods already described in preceding demos. Line 10 invokes the member function `Map::Inverse()` to load the inverse of `f` into `u`. We decide to retain terms only through degree 4.

Lines 11-14: The coefficients in `f` and `u` are printed first. Each is a three-component map, each component being a Jet. The components are written separately and identified. In comparing the program variable `f` to the mathematical function, f, remember to take the reference point into account.

Lines 15-18: Here is the acid test. If indeed `f` and `u` model inverse Jets, then it must be that `f(u)` and `u(f)` model the identity functions at their respective reference points, $\underline{u}_o$ and $f(\underline{u}_o)$. The output shows that this is indeed the case apart from a term of fourth degree in `u(f)` which is clearly due to machine error. *You should thoroughly review the output at this point to be sure you understand why the printed coefficients support this claim.*

We have not yet included a program using complex valued jets. So as to include at least one such example, what follows is source code for an analogous inverse calculation but using complex maps with a reference point of $(-1.2+0.9i, 3.5+1.7i, 2.1-0.3i)$. The required extra call to **JetC::Setup** is a flaw in the MXYZPTLK software and will be corrected. The rest of the program is just written in parallel to the previous one, except that **coordC** and **CMap** objects are used.

Source:

```
 1  #include "mxyzptlk.rsc"

 2  main( ) {
 3
 4   Jet::Setup ( 3, 4, 3 );
 5   JetC::Setup( 3, 4, 3 );

 6   coordC  x( complex( -1.2,  0.9 ) ),
 7           y( complex(  3.5,  1.7 ) ),
 8           z( complex(  2.1, -0.3 ) );
 9   CMap    w, u;

10   w.SetComponent( 0, complex(  3.0,  1.0 ) + x + 3.0*y + x*y - y*z   );
11   w.SetComponent( 1, complex( -1.0,  0.2 ) + y - x + z + x*z + y*y   );
12   w.SetComponent( 2, complex(  2.0, -0.9 ) + z + 2.0*x + y*z - x*y*z );
```

```
 u = w.Inverse();

 cout << "\n====== w.printCoeffs(); ===================\n" << endl;
 w.printCoeffs();
 cout << "\n====== u.printCoeffs(); ===================\n" << endl;
 u.printCoeffs();
 cout << "\n====== w(u).printCoeffs(); ================\n" << endl;
 w(u).printCoeffs();
 cout << "\n====== u(w).printCoeffs(); ================\n" << endl;
 u(w).printCoeffs();
}
```

3.9 Lie Operators

MXYZPTLK contains a Lie operator object which acts on Jets. In the example below, we will model the "equations of motion,"

$$\dot{x} = x(2y^3 - x^3), \quad \dot{y} = -y(2x^3 - y^3) \quad .$$

using the Lie operator,

$$\mathbf{V} = x(2y^3 - x^3)\frac{\partial}{\partial x} - y(2x^3 - y^3)\frac{\partial}{\partial y} \quad . \tag{12}$$

This vector field possesses an invariant: $f(x, y) = x^2/y + y^2/x$. The program will test the invariant property by applying the Lie operator. That is, it will check the condition, $\mathbf{V}f = 0$.

Source: Lie_K_Test.cc

```
#include "mxyzptlk.rsc"

main( int argc, char** argv ) {

  if( argc != 4 ) {
  cout << "\nUsage: " << argv[0]
       << "  deg  x  y"
       << endl;
  exit(0);
 }

 Jet::Setup( 2, atoi( argv[1] ), 2 );
 coord  x( atof( argv[2] ) ),  y( atof( argv[3] ) );
11   LieOperator V;
12   V.SetComponent( 0,   x*( 2.0*pow( y, 3 ) - pow( x, 3 ) ) );
13   V.SetComponent( 1, - y*( 2.0*pow( x, 3 ) - pow( y, 3 ) ) );

14   ( V ^ ( x*x/y + y*y/x ) ).printCoeffs();
15  }
```

Output:

```
hazel 1: Lie\_K\_Test 5 1.7 3.5

Count  = 6, Weight = 5, Max accurate weight = 4
Reference point:
1.700000e+00  3.500000e+00
```

Comments:

Lines 11-13: After declaring the `LieOperator V` we set its components in the same way as we would a `Map` variable. In fact, the two are, more or less, synonomous, in the sense that both contain an array of jets. Of course, they are very different mathematical objects.

Line 14: The mathematical operation of a Lie operator on a function is implemented via the method **LieOperator::operator^**`}( const Jet\& ).} The operator symbol ‘‘*`” was not used for this in order to avoid confusion with statements like,

```
LieOperator V, W, Y;
Jet f;
...
W = f*V;
Y = V*f;
~~,
```

which are meant to model the mathematical operations, $\mathbf{W} = \mathbf{Y} = f\mathbf{V}$. The output shows no terms because the only non-zero ones are due to machine error appearing at degree 5, which is higher than the "Max accurate weight."[5] That no terms are printed is equivalent to saying that the Jet is identically zero.

Of course, this test applies only to the reference point. *Automatic differentiation is not symbolic differentiation,* for which a zero result would apply everywhere.

[5]The fact that these useless terms are even carried around is an anomaly that, it is hoped, will be eliminated in future versions of MXYZPTLK.

3.10 Brackets

The commutator of two Lie operators is itself a Lie operator. This binary operation is accomplished in MXYZPTLK by sandwiching the operator symbol "^" between two `LieOperator` objects. We will illustrate its use by calculating the action of **V**, defined in Eq.(12), **W**, defined as

$$\mathbf{W} = y\frac{\partial}{\partial x} - x\frac{\partial}{\partial y} \quad ,$$

and $[\,\mathbf{V},\, \mathbf{W}\,]$ on the function $f = x^2 + y^2$.

Source: lbtest.cc

```
#include "mxyzptlk.rsc"

main( int argc, char** argv ) {

  if( argc != 4 ) {
  cout << "\nUsage: " << argv[0]
       << " deg  x  y"
       << endl;
  exit(0);
 }

 Jet::Setup( 2, atoi( argv[1] ), 2 );
 coord  x( atof( argv[2] ) ),  y( atof( argv[3] ) );
 Jet    f = x*x + y*y;        // Equivalent to "Jet f( x*x + y*y );"

 LieOperator V, W;
 V.SetComponent( 0,   x*( 2.0*pow( y, 3 ) - pow( x, 3 ) ) );
 V.SetComponent( 1, - y*( 2.0*pow( x, 3 ) - pow( y, 3 ) ) );
 W.SetComponent( 0,  y );
 W.SetComponent( 1, -x );

 ( V^f )     .printCoeffs();
 ( W^f )     .printCoeffs();
 ( (V^W)^f ) .printCoeffs();

}
```

Output:

```
hazel 1: lbtest 3 1 1

Count  = 8, Weight = 3, Max accurate weight = 2
```

```
Reference point:
1.000000e+00  1.000000e+00

Index:    0  1   Value:   1.400000e+01
Index:    1  0   Value:  -1.400000e+01
Index:    0  2   Value:   2.800000e+01
Index:    2  0   Value:  -2.800000e+01

Count  = 0, Weight = -1, Max accurate weight = 2
Reference point:
1.000000e+00  1.000000e+00

Count  = 10, Weight = 3, Max accurate weight = 2
Reference point:
1.000000e+00  1.000000e+00

Index:    0  0   Value:   2.800000e+01
Index:    0  1   Value:   7.000000e+01
Index:    1  0   Value:   7.000000e+01
Index:    0  2   Value:   6.000000e+01
Index:    1  1   Value:   1.600000e+02
Index:    2  0   Value:   6.000000e+01
```

Comments:

Line 11: This is the declaration of a `Jet` variable using its copy constructor. **BE AWARE:** doing this is a little dangerous, because `Jet`s employ an envelope-letter idiom for storing data. It is recommended that `Jet` variables always be declared and initialized separately. In this case, it would have been better to have written,

```
    Jet f;
    f = x*x + y*y;
```

Line 19: Most of the program is similar to what has gone before. This line contains the only new operation, taking the commutator of `V` and `W` before acting on `f`.

MXYZPTLK also contains a Poisson bracket operation, accomplished by sandwiching the operator symbol "^" between two Jets. That is, `f^g` models $\{ f, g \}$ when `f` and `g` are `Jet` variables just as `U^V` models $[U, V]$ when `U` and `V` are `LieOperator` variables. The next example employs both Lie brackets and Poisson brackets to test the well known antimorphism,

$$[\mathbf{V}_a, \mathbf{V}_b] = -\mathbf{V}_{\{a,b\}} \quad .$$

We will let

$$\begin{aligned} a(\underline{x}, \underline{p}) &= x_1^2 x_2^3 p_1 p_2^4 \ , \\ b(\underline{x}, \underline{p}) &= \sin(x_1 p_2^2 x_2^3) \ . \end{aligned}$$

The brackets will be evaluated at the arbitrarily selected reference point, $(\underline{x}, \underline{p}) = (0.32, 0.5, -3.1, 1.5)$. As an added bonus, we will test the Jacobi identity, using a third function, $c = \exp(p_1 x_1 + p_2 x_2)$.

Source:

```
#include "mxyzptlk.rsc"

main() {
 double u1( 0.32 ), u2( 0.5 ),
        v1( -3.1 ), v2( 1.5 );

 Jet::Setup( 4, 6, 4 );

 double w, y, z, answer;
 coord  x1( u1 ), x2( u2 ), p1( v1 ), p2( v2 );
 Jet    a, b, c, pb;

 // -- Calculation of Poisson bracket via Jets
 a = (x1*x1) * (x2*x2*x2) * p1 * (p2*p2*p2*p2);
 b = sin( x1 * (p2*p2) * (x2*x2*x2) );
 pb = a^b;
 cout << "Computed by Jet: " << pb.standardPart() << "\n";

 // -- Hand calculations
 w = (u1*u1) * (u2*u2*u2) * v1 * (v2*v2*v2*v2);
 y =      u1 * (v2*v2) * (u2*u2*u2)  ;
 z = cos( y );
 answer = w*y*z*( 6.0/(u2*v2) - 1.0/(u1*v1) - 12.0/(u2*v2) );
 cout << "Exact answer   : " << answer << "\n";

 cout << "And also       : "
      << ( ( (x1*x1) * (x2*x2*x2) * p1 * (p2*p2*p2*p2) ) ^
           ( sin( x1 * (p2*p2) * (x2*x2*x2) ) )
         ).standardPart()
      << "\n\n";

 // -- Test of the Jacobi identity
 c = exp( p1*x1 + p2*x2 );
 cout << "Jacobi identity" << endl;
 ( (a^(b^c)) + (b^(c^a)) + (c^(a^b)) ).printCoeffs();

 // -- Hamiltonian vector fields
 LieOperator V_a ( a  );
 LieOperator V_b ( b  );
 LieOperator V_pb( pb );

 cout << "Hamiltonian test" << endl;
 ( V_pb + ( V_a ^ V_b ) ).printCoeffs();
}
```

Output:

```
hazel 2: pbtest
Computed by Jet: 0.125897
Exact answer   : 0.125897
And also       : 0.125897

Jacobi identity

Count  = 84, Weight = 6, Max accurate weight = 4
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00

Hamiltonian test

************ Begin LieOperator::printCoeffs ********
Weight: 6

******************
**** Component index = 0

Count  = 28, Weight = 6, Max accurate weight = 4
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00

******************
**** Component index = 1

Count  = 49, Weight = 6, Max accurate weight = 4
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00

******************
**** Component index = 2

Count  = 49, Weight = 6, Max accurate weight = 4
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00
```

```
******************
**** Component index = 3

Count  = 49, Weight = 6, Max accurate weight = 4
Reference point:
3.200000e-01  5.000000e-01  -3.100000e+00  1.500000e+00

************ End   LieOperator::printCoeffs ********
```

Comments:

Lines 9-12, 14-18: The Poisson bracket is computed two ways: (1) using the binary operator `^` on Jet variables `a` and `b` and (2) for comparison, using its algebraic expansion on variables of type double. Line 12 contains the actual Poisson bracket, written as a binary operator on two `Jet` variables, using the same symbol as for `LieOperator` variables.

Lines 20-24: This third calculation emphasizes that Jet methods and operators work not only on formally declared Jet variables but also on expressions which evaluate to Jet variables. Of course, that is obtained for free as a feature of the C++ language.. The hand calculation of Lines 14-18 is repeated but using `coords`.

Line 28: This tests the Jacobi identity. The expression should evaluate to zero, and the **Output** indicates that it does. The extra parentheses make certain that everything gets evaluated in the proper order. Not only is the Poisson bracket operation non-associative (and non-commutative), its precedence relative to other operations is an issue best left unexplored.

Lines 30-32: This form of declaring a `LieOperator` takes a `Jet` variable as an argument and builds the Hamiltonian vector field associated with it.

Line 34: Finally, the morphism test itself. What is calculated here is,

$$\mathbf{V}_{\{\,a,\,b\,\}} + [\,\mathbf{V}_a,\,\mathbf{V}_b\,] \quad ,$$

and the **Output** indicates that the result is indeed zero. (That is, no terms are printed.)

3.11 Exponential maps

The member function **LieOperator::expMap** performs an exponential map of a LieOperator and applies it to a `Jet` to obtain the resulting `Jet`. In the example below, we will use **LieOperator::expMap** to "integrate" the equations of motion,

$$\dot{x} = x(2y^3 - x^3), \quad \dot{y} = -y(2x^3 - y^3) \quad .$$

using the Lie operator already written in Eq.(12). Recall that this vector field possesses an invariant: $x^2/y + y^2/x$. The program tests the map by the value of this invariant both before and after the time step.

Source: Lie_L_Test.cc

```
#include "mxyzptlk.rsc"

main( int argc, char** argv ) {

  if( argc != 3 ) {
  cout << "\nUsage: " << argv[0]
       << "  deg  t"
       << endl;
  exit(0);
 }

 int    deg = atoi( argv[1] );
 double   t = atof( argv[2] );
 Jet::Setup( 2, deg, 2 );
 coord  x( 0.0 ),  y( 0.0 );

 LieOperator V;
 V.SetComponent( 0,   x*( 2.0*pow( y, 3 ) - pow( x, 3 ) ) );
 V.SetComponent( 1, - y*( 2.0*pow( x, 3 ) - pow( y, 3 ) ) );

 Jet f, g;
 f = V.expMap( t, x );
 g = V.expMap( t, y );

 double a, b, z[2];
 while(1) {
   cout << "Enter x and y: ";
   cin  >> z[0] >> z[1];
   a = f( z );
   b = g( z );
   cout << "( " << z[0] << ", " << z[1] << " ) maps to ( "
                << a    << ", " << b    << " )" << endl;
   cout <<    "Before: " << setprecision(5)
        << z[0]*z[0]/z[1] + z[1]*z[1]/z[0]
        << "   After:  " << setprecision(5)
        << a*a/b + b*b/a << endl;
 }

}
```

Output:

```
hazel 1: Lie\_L\_Test 20 1.
Enter x and y: .3 .5
```

```
( 0.3, 0.5 ) maps to ( 0.380158, 0.53075 )
Before: 1.0133   After:  1.0133
Enter x and y: .4 .6
( 0.4, 0.6 ) maps to ( 0.56134, 0.59739 )
Before: 1.1667   After:  1.1632
Enter x and y: .5 .7
( 0.5, 0.7 ) maps to ( 0.65366, 0.57497 )
Before: 1.3371   After:  1.2489
Enter x and y: .6 .8
( 0.6, 0.8 ) maps to ( 0.75124, 1.4679 )
Before: 1.5167   After:  3.2528
Enter x and y: ^C

hazel 2: Lie\_L\_Test 20 -1.
Enter x and y: -.3 -.5
( -0.3, -0.5 ) maps to ( -0.380158, -0.53075 )
Before: -1.0133   After:  -1.0133
Enter x and y: -.4 -.6
( -0.4, -0.6 ) maps to ( -0.56134, -0.59739 )
Before: -1.1667   After:  -1.1632
Enter x and y: -.5 -.7
( -0.5, -0.7 ) maps to ( -0.65366, -0.57497 )
Before: -1.3371   After:  -1.2489
Enter x and y: -.6 -.8
( -0.6, -0.8 ) maps to ( -0.75124, -1.4679 )
Before: -1.5167   After:  -3.2528
Enter x and y: ^C
```

Comments:

Lines 9-12: The degree of the representative polynomial is established, and the size of the time step is read from the command line. After **Jet::Setup** requests a two-dimensional phase space, its coordinates, `x` and `y`, are declared.

Lines 13-15: Components of the `LieOperator V` is constructed so as to model the vector field, **V**, written above.

Lines 17-18: These lines perform the exponential map operation on the jets `x` and `y`. The corresponding mathematical operation would be written,

$$f = e^{t\mathbf{V}}x, \quad g = e^{t\mathbf{V}}y \quad .$$

We are applying the exponential map the coordinate functions themselves. Thus, if $(x_1, y_1)/mapsto(x_2, y_2)$ under the flow of Eq.(12), it must be that $x_2 = f(x_1)$ and $y_2 = g(y_1)$.

Lines 19-31: Within an indefinite loop, points are entered and converted with Jets `f` and `g`. Values of the invariant are printed for the "initial" and "final" states. The **Output** shows two runs of this program, for time steps ± 1. Polynomials are truncated at degree 20. There are two things to note: (a) symmetry is correctly preserved (i.e., $t \to -t$, $x \to -x$, and $y \to -y$), and (b) the polynomial representation fails rapidly as the size of the argument increases. The latter property is not helped by taking more polynomial terms. The problem of determining the radius of convergence of an exponential map is an ongoing topic of research. Note, however, that regardless of whether the series converges or not, *the coefficients in the truncated polynomial are computed exactly.*[6]

[6] There is a codicil to this: $\mathbf{V}$ must map zero to zero and not start with a linear term.

4 Functions and methods

In this section we describe the functions and methods[7] currently available in MXYZPTLK, arranged in the order in which they probably would be used in most programs.

4.1 Setup function

void Jet::Setup(int n, int w, int s, double* r, double* sc)

Before Jet variables can be used, the application program must provide information on the dimensions of the problem space and on an initial reference point. This is done with a **Setup** function which must be invoked before using Jet variables in arithmetic or analytic operations. The formal arguments, all input, are interpreted as follows.

int n: Dimension of the problem space, the total number of dynamical and control coordinates.

int w: The maximum derivative weight to be carried by Jet variables. If we interpret a Jet variable as a multinomial, then its degree will be $\leq$ **w**.

int s: The number of dynamical coordinates, i.e., the dimension of "phase space."

double r[n]: An array containing the reference point.

double sc[n]: An array containing numbers characterizing the scale of each coordinate.

Every argument is provided a default value in the header file Jet.hxx. These are: n = 6, w = 1, s = 0, r = 0, and sc = 0. If **s** is not declared explicitly, the default option of 0 means that all variables are considered to be control variables, and neither concatenation nor Poisson brackets will be allowed (see Sections 4.7 and 4.8). If a reference point is not declared, it will be set to the "origin," an array of zeroes. Finally, if the scaling array, sc, is not explicitly given, **Jet::Setup** will assume that all the values of all variables will have roughly unit magnitude. **Jet::Setup** will stop the application program if arguments **s** and **n** do not satisfy $0 \leq$ **s** $\leq$ **n**.

In principle, **Jet::Setup** should be invoked before the formal declaration of Jet variables, but this is not always possible. For example, an application program may contain a fragment like this:

[7] A "method" is a public member function of either the Jet or LieOperator class.

```
Jet x;
Jet y;

main() {
  Jet::Setup();
  ...
}.
```

Here, `x` and `y` are meant to be global variables, so they are initialized when the program begins to run and *before* the **Jet::Setup** function can be invoked. What happens in such a case is this: the C++ Jet constructors only *partially* initialize these variables and load their addresses into a queue. When **Jet::Setup** is finally invoked, this queue is traversed, and the initialization of any variable which had been declared previously is completed.

4.2 Setting the reference point

(a)	**static**	**void Jet::FixReference**	**(const double*)**		
(b)	**static**	**void Jet::FixReference**	**(const int*)**		
(c)	**static**	**void Jet::FixReference**	**(const Jet&)**		
(d)	**static**	**void Jet::FixReferenceAtStart**	**(const LieOperator&)**		
(e)	**static**	**void Jet::FixReferenceAtEnd**	**(const LieOperator&)**		
(f)		**void Jet::fixReference**	**()**		
(g)		**void Jet::fixReference**	**(const double*)**		
(h)		**void Jet::fixReference**	**(const Jet&)**		
(i)		**void Jet::fixReferenceAtEnd**	**(const T&)**		
(j)		**void Jet::fixReferenceAtStart**	**(const T&)**		
(k)		**void T ::fixReference**	**(double*)**	**Note:**	**T is either a Map**
(l)		**void T ::fixReference**	**()**		**or a LieOperator**
(m)		**void T ::fixReference**	**(Jet&)**		
(n)		**void T ::fixReferenceAtEnd**	**(const T&)**		
(o)		**void T ::fixReferenceAtStart**	**(const T&)**		

Every `Jet` variable carries the coefficients of a polynomial that is the simplest representative of an equivalence class of functions. In addition, it also carries the reference point at which the equivalence class is established. Whenever a `Jet` variable is declared, therefore, a reference point must be given. If one is not assigned explicitly, it is done implicitly by using a default reference point, established initially as the argument of a **Jet::Setup** fuction. `Jet` variables declared either before or after its invocation are assigned this reference point as their own. Alternatively, if the calculation is initialized by declaring a number of `coord` variables, then their values automatically become the components of the default reference point.

However, the default reference point need not remain the same throughout a program. It can be changed by one of the first four functions listed above. The first sets it value to that of an array provided by the user. Changing this array later in the application program will not, by itself, change the default reference; another invocation of **Jet::FixReference** would be required. Form (b) of this function sets (or resets) the default reference point to that of an already defined Jet variable. The third function, **Jet::FixReferenceAtStart**, sets the default reference to the reference point of its argument; the fourth, **Jet::FixReferenceAtEnd**, sets it to the *standard part* of its argument. For example, suppose the first component of a LieOperator `u` prolongs the function $\cos(xy + \pi/2)$, while the second component prolongs $\sin(xy + \pi/2)$, both about the point $(x, y) = (\sqrt{\pi}, -\sqrt{\pi})$. Then "`Jet::FixReferenceAtStart( u )`" would set the default reference to $(\sqrt{\pi}, -\sqrt{\pi})$, while "`Jet::FixReferenceAtEnd( u )`" would set it to $(0, -1)$. The latter function is essential for doing concatenation correctly (see Sections 4.7 and 3.7).

The ten methods (e)-(n) are public members of the Jet and LieOperator classes. They perform analagously to the first four, but rather than acting on the default reference point, these members adjust the reference point of the individual variables. For example, in the fragment

```
Jet x, y, z;
...
x.fixReference( y );
z.fixReference();
```

the **.fixReference** member sets the reference point of `x` to that of `y`, while the reference point of `z` is set to the current default reference.

4.3 Initializing a calculation: coordinates

(a)	**coord::coord**	**(double x)**		
(b) void Jet	**::setVariable**	**(int j)**		
(c) void Jet	**::setVariable**	**(double x, int j)**		
(d) void T	**::SetComponent**	**(int j, const Jet& x)**	**Note:**	**T is either a Map or a LieOperator**

AD/DA arithmetic must begin by identifying a set of variables as differentiable coordinate functions. The simplest way of doing this is to declare **coord** variables, as was done in the demos of Section 3. However, this is not the only way. Jet variables can also act like coordinates After setting the default reference point with **Jet::Setup** or **JetFixReference**, one simply assigns an "index" to each coordinate variable, as in the fragment below.

```
...
static double r[] = { 0., 1., -1. };
Jet::Setup( 3, 12, 0, r );

Jet x, y, z, f;

x.setVariable( 0 );
y.setVariable( 1 );
z.setVariable( 2 );

f = exp( x*y + z );
```

This identifies the phase space coordinate array, $\underline{u} \equiv (x, y, z)$. The variable `f` will contain data on the differentiable function, $f(\underline{u}) = e^{xy+z}$, with derivatives evaluated at the point $\underline{u} = (0, 1, -1)$. These data can be accessed through a selection method (explained in Section 4.6) by using the indices that were assigned by **.setVariable**.

A second way of initializing a Jet calculation employs the second form of **.setVariable** to declare a Jet variable as a coordinate while simultaneously setting its value. This method is not recommended: it resets the default reference point one component at a time, so that a invoking **.fixReference** would be required after the fact.

```
...
Jet::Setup( 3, 12 );
Jet x, y, z, f;

x.setVariable( 0.0, 0 );
y.setVariable( 1.0, 1 );
z.setVariable( -1.0, 2 );

x.fixReference();
y.fixReference();

f = exp( x*y + z );
```

The two LieOperator methods enable one to declare a component of a LieOperator variable to be a coordinate — which is useful in the control sector — or to load Jet variables into specific components — prior to concatenation, for example. Their use was illustrated in Section 3.

4.4 Operators

Logical and arithmetic binary operators act the way one naturally expects. The replacement operator, `=`, enables the replacement of one Jet, or LieOperator, variable by another, while the logical operators `==` and `!=` test whether two variables are equivalent. Arithmetic operators `+`, `-`, `*` and `/`, when sandwiched between two Jet variables, activate the corresponding arithmetic operations of addition, subtraction, multiplication, and division. In addition, the subtraction symbol, `-`, also acts as a unary operator on Jet variables, indicating that they are to be negated. The C++ operators `+=`, `-=`, `*=`, and `/=` are available as well.

When placed between two LieOperator variables, the "multiplication" operator, `*`, initiates concatenation rather than multiplication. This will be discussed in detail in Section 4.7.

Components of a LieOperator can be accessed as one would expect, using member function **Jet LieOperator::operator()(int)**. For example,

```
Jet x, y, z, ... ;
LieOperator u;
...
x = u(0);
y = u(1);
z = u(2);
...
```

will load the zero-th component of `u` into `x`, the first into `y`, and so forth.

In addition to these, the binary operator caret, `^`, placed between two `LieOperator`s takes their commutator, and between two `Jet`s, performs a Poisson bracket. We delay its description to Section 4.8.

All binary operators *except concatenation*, which has its own subtleties, check to be sure that their two operands have the same reference point. If they do not, then an error message is written on the standard output, and the application program is stopped. Of course, the replacement operator, `=`, automatically sets the reference point of its left-hand operand to that of the right-hand one.

4.5 Transcendental functions

Most of the C++ transcendental functions available for "`double`" variables have been written for `Jet` variables as well. Specifically, the MXYZPTLK library currently contains the functions sin, cos, tan, asin, acos, atan, exp, sinh, cosh, tanh, log, log10, pow, sqrt, and w (the complex error function). Except for pow, each takes a **Jet** argument and, as one would expect, returns a **Jet** result. Two signatures are available for pow: `Jet pow( const Jet&, int )` and `Jet pow( const Jet&, double )`.

4.6 Selection methods

(a) double	**Jet::standardPart**	**()**		
(b) double	**Jet::derivative**	**(int* m)**		
(c) double	**Jet::weightedDerivative**	**(int* m)**		
(d) void	**T ::standardPart**	**(double* x)**		**Note: T is either a Map**
(e) void	**T ::derivative**	**(int* m, double* x)**		**or a LieOperator.**
(f) void	**T ::weightedDerivative**	**(int* m, double* x)**		**W is either a Jet,**
(g) W	**W ::filter**	**(int wgtLo, int wgtHi)**		**a Map, or a LieOperator.**
(h) W	**W ::filter**	**(char (*f) (const int*, double) [])**		

A number of methods access parts of Jet variables without changing the variable. As a `Jet` member function, **.standardPart**, returns as its value the image of the reference point, $f(\underline{u}_o)$.[8] As a LieOperator method, it accepts an array pointer (that is, the name of an array) as argument and loads the "standard parts" of all its components into this array. For example:

```
...
double       x[8];
LieOperator  y;
...
y.standartPart( x );
if( x[3] == y(3).standartPart() ) cout << "All is OK\n"
...
```

The **.derivative** and **.weightedDerivative** routines return the value of a specified derivative or component of the polynomial representative of the jet. Their argument is interpreted as the name of an integer array containing the indices of the desired derivative. For example, if `f` models a jet containing $f : R^3 \to R$, at the point $\underline{w}$, then the derivative $\partial^6 f(\underline{x})/\partial x_0 \partial x_1^3 \partial x_2^2|_{\underline{x}=\underline{w}}$ can be obtained as follows.

```
...
Jet        f;
double     d, w[3];
static int m[] = { 1, 3, 2 };
...
Jet::Setup( 3, 10, w );
...
d = f.derivative( m );
...
```

The **.weightedDerivative** returns a polynomial coefficient, which is the derivative weighted by factorials of the indices. These are, according to Eq.(2), the actual coefficients which would appear in the truncated poly-

[8]The name of this method is a throwback to the days when connections between DA and nonstandard analysis were being stressed.

nomial representation of f, and they, not the derivatives, are the actual numbers stored in a `Jet` variable.[9] Thus, if we replace `.derivative` with `.weightedDerivative` in the example above, then the value returned would be $(1!\,3!\,2!)^{-1}\partial^6 f(\underline{x})/\partial x_0 \partial x_1^3 \partial x_2^2|_{\underline{x}=\underline{w}}$.

As with **.standardPart**, the `LieOperator` and `Map` versions of **.derivative** and **.weightedDerivative** load the values of the derivative for each component of the operator (or map) into the array pointed to by the additional argument, **double* x**.

The **.filter** methods return a variable whose polynomial terms are a subset of those of the object on which they are invoked. Letting **W** stand for either a `Jet`, `Map`, or `LieOperator`, form (g) returns a **W** object with terms whose degrees are bounded by the arguments, **wgtLo** and **wgtHi**, inclusively. Form (h) is more flexible, taking as its argument an array of decision functions which determine the terms to be filtered into the **W** object to be returned. As an example, consider the code fragment,

```
char c0( const int* index, double  /* value */ ) {
  return  index[0] == 0;
}
char c1( const int* index, double /* value */ ) {
  return index[0] == 0 && index[1] < 5;
}
char c2( const int* index, const complex /* value */ ) {
  return value > 100.0;
}

typedef char (*FUNCPTR)(const int*, const complex);
static FUNCPTR crit[] = { c0, c1, c2 };

main() {
...
Map f, g;
...
g = f.filter( crit );
...
}
```

The first argument of the criterion functions is interpreted as an array of integers which represent the index of one term in a polynomial; the second argument is interpreted as the value of the coefficient associated with the first argument. Given this information, the function decides whether the term passes the filter. With maps, different filters can act on different components of the map, which is the reason for putting them into an array. In the fragment above, because of their positions in the array `crit`, `c0` examines the terms in

[9] In fact, the **.derivative** method first invokes **.weightedDerivative** and then multiplies by the factorials.

`f(0)`, `c1` in `f(1)`, and `c2` in `f(2)`. Acting on

$$f : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} 300xy^2 - 12y^3 + yz \\ 32y^8z^4 + 72y^2 \\ 137x^2y^2 - 75xyz \end{pmatrix}$$

it would produce the result

$$g : \begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} -12y^3 + yz \\ 72y^2 \\ 137x^2y^2 \end{pmatrix} \quad .$$

4.7 Evaluation and concatenation

(a) double	**Jet::operator() (double*)**		
(b) Jet	**Jet::operator() (Jet*)**		
(c) Jet	**Jet::operator() (T&)**	**Note: T is either a Map**	
(d) T	**T ::operator() (T&)**	**or a LieOperator.**	

A `Jet` variable stores the coefficients of a truncated polynomial. Form (a) above enables one to evaluate that polynomial at a point in the problem space. The argument is interpreted as an array of `doubles` containing the point of evaluation. `Jets` and `Maps` keep track of their own reference points, so that the user program need not subtract it explicitly in specifying the argument. (See the example in Section 3.5.)

Let $\underline{\phi}, \underline{\psi} : R^{N_d+N_c} \rightarrow R^{N_d+N_c}$ be two mappings of the problem space into itself *which act like the identity on the control sector.* That is, only the dynamical coordinates change under the action of $\underline{\phi}$ and $\underline{\psi}$; the control variables are not touched. The composite map, $\underline{h} = \underline{\phi} \circ \underline{\psi} : \underline{u} \mapsto \underline{\phi}(\underline{\psi}(\underline{u}))$, is a mapping of the same type. This operation is performed by form (d) of **.operator()**. However, notice that although the reference points of $\underline{h}$ and $\underline{\psi}$ are identical, say $\underline{a}$, the reference point of $\underline{\phi}$ is $\underline{\psi}(\underline{a})$. The reference point must be explicitly declared, using the **fixReference** methods, before performing concatenation.[10] The demo in Section 3.7 provides an example showing how this is done.

When the control sector is not empty, *all* `Map` *and* `LieOperator` *operations and methods assume that the first N_d components refer to the dynamical sector and the final N_c to the control sector,* these having been determined by the **Jet::Setup** function.

Keep in mind that all manipulations are performed on *truncated* polynomials. Thus, $\underline{\phi} \circ \underline{\psi}$ will contain terms only up to the degree of truncation. All higher degree terms which normally appear when concatenating two polynomials are ruthlessly eliminated.

[10] What physicists call "concatenation," mathematicians call "composition."

Form (c) is similar except that a single jet is concatenated with a map. Thus, if $f, g : R^N \to R$ and $\underline{\phi} : R^N \to R^N$, then the correspondence is:

$$g = f \circ \underline{\phi} \quad \leftrightarrow \quad \texttt{g = f(Phi);} \quad ,$$

where `g` and `f` are `Jet` variables, and `Phi` is a `Map` variable.

Form (b) of concatenation will work only if the problem space is one dimensional, for the operation $f \circ g$ does not make sense otherwise. Similarly to form (a), the argument is interpreted as an array of `Jet` variables.

4.8 Differentiation and Poisson brackets

(a) Jet	**Jet**	**::D**	**(int* m)**
(b) Jet	**LieOperator**	**::operator^**	**(Jet&)**
(c) LieOperator		**operator^**	**(LieOperator&, LieOperator&)**
(d) Jet		**operator^**	**(Jet& x, Jet& y)**

Derivatives of jets are themselves jets, and each function listed above performs an action of differentiation. For example, if $u, v : R^5 \to R$, and we want to implement the functional correspondence, $v \equiv \partial^7 u / \partial x_0^2 \partial x_1 \partial x_3^4$, using `Jet` variables, this could be accomplished as follows.

```
...
Jet::Setup( 5, 10 );
Jet u, v;
static int m[] { 2, 1, 0, 4, 0 };
...
v = u.D( m );
```

The Jet variable `u` itself would be unchanged by this method.

Taking derivatives lower the degree of a mathematical jet and, correspondingly, *lowers the maximum accurate weight* of a Jet variable. Thus, if `u` stores derivatives of the real valued function u through weight w, and we define v to be an m^{th}-order derivative of u, then `v` can store the derivatives of v accurately only through weight $w - m$, all derivatives of higher weight being unknown. In the small fragment shown above, only derivatives through order 3 will be correctly stored in `v`, because the call `Jet::Setup` requested only derivaties through order maximum order 10 be stored in any `Jet` variable, particularly, in `u`. *A private datum of each Jet variable keeps track of the maximum weight of accurately stored derivatives,* which may be less than the maximum weight declared by the **Jet::Setup** function. These data are used by and propagated through arithmetic operations, so that errors will not arise. In principle, an applications program could request a differentiation or invoke a selection method which cannot be carried out accurately because of differentiations executed previously. If this happens, then the Jet class will refuse to cooperate and will write an error message to the standard output.

The badly overloaded operator symbol `^` indicates the action of a `LieOperator` on a `Jet` and two different kinds of brackets. Form (b) implements the former.

$$g = \mathbf{V} f \quad \leftrightarrow \quad \texttt{g = V^f}$$

Form (c) implements the commutator of two Lie operators, which is itself a Lie operator.

$$\mathbf{U} = [\,\mathbf{V}, \mathbf{W}\,] \quad \leftrightarrow \quad \texttt{U = V^W}$$

Finally, if the dynamical sector has even dimension, say $N_d = 2n$, it can be (and usually is) interpreted as a phase space whose first n components are "positions" and whose second are "momenta." The Poisson bracket is then well defined, and Jet implements this operation via form (d).

$$h = \{\, f, g \,\} \quad \leftrightarrow \quad \texttt{h = f^g}$$

Because all these operation requires taking derivatives, the maximum accurate weight of the resultant is generally smaller than that of its operands. This reduction does not occur, however, when the objects map their reference points to the origin, for example, if the image of the reference point is zero under both `f` and `g`. Again, MXYZPTLK stores that information automatically so that the user program need not keep track of it explicitly.

ACKNOWLEDGEMENT

I was indebted to Herbert Wilf for a stimulating discussion of his ranking algorithm [14, 15, 9] which is at the core of evaluation and concatenation. Jim Holt extended the code for classes **Jet** and **LieOperator** to implement classes **JetC** and **CLieOperator**. I am also grateful to him for using MXYZPTLK in his own programs. His continually pushing this library to its limits has been repeatedly helpful in exposing weak points and suggesting improvements.

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] M. Berz. Differential algebra – a new tool. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference.* IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

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

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

[5] Bruce Eckel. *Using C++* . Osborne McGraw-Hill, Berkeley, 1989.

[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), March 1989. To be published.

[7] Leo Michelotti. Differential algebras without differentials: an easy C++ implementation. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

[8] Leo Michelotti. Exploratory orbit analysis. In Floyd Bennett and Joyce Kopta, editors, *Proceedings of the 1989 IEEE Particle Accelerator Conference*. IEEE, March 20-23, 1989. IEEE Catalog Number 89CH2669-0.

[9] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial Algorithms*. Academic Press, New York, 1978.

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

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

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

[13] Roy Thatcher. Programming in C - a word of caution. Fermilab Computing Division Newsletter, Vol. XVIII, No. 1, pp. 3-4, Jan-Feb 1990.

[14] Herbert S. Wilf. A unified setting for sequencing, ranking, and selection algorithms for combinatorial objects. *Advances in Mathematics*, 24:281–291, 1977.

[15] Herbert S. Wilf. A unified setting for selection algorithms (II). *Annals of Discrete Mathematics*, 2:135–148, 1978.