Automatic Differentiation of Limit Functions

Leo Michelotti
Fermilab*, P.O.Box 500, Batavia, IL 60510

Abstract

Automatic differentiation can be used to evaluate the derivatives of and set up Taylor series for implicitly defined functions and maps. We provide several examples of how this works, within the context of the MXYZPTLK class library, and discuss its extension to inverse functions.

I. Introduction.

The techniques of automatic differentiation [2] and differential algebra [1, 3, 9] are rapidly becoming a standard part of accelerator physicists' arsenals. That automatic differentiation can be used to calculate the derivatives of recursively or iteratively[1] defined functions is not as well appreciated as it should be. Applying recursive algorithms directly to `DA` variables[2] provides an easy method for obtaining derivatives of such functions. In the following sections we shall (a) sketch the essential argument needed to prove this assertion, (b) discuss two examples written using the C++ classes in MXYZPTLK [5, 6], and (c) provide C++ code fragments for a general program. This paper builds on work done previously [7] but remains necessarily short. Applications of the techniques described are too numerous to mention and so obvious that there is no need to do so.

II. Heuristics for a proof

Following [7], a mathematically correct proof that recursions can be extended to `DA` variables was published by Gilbert [4] for single derivatives; its extension to higher derivatives is implied. The proof's correctness makes it formal, and how it works is not obvious to naive intuitionists. It is hoped that the following heuristic argument will be easier to understand while retaining essential points of the proof.

Let $f : R \rightarrow R$ possess a fixed point, x^*. The sequence $x_{n+1} = f(x_n)$, started "close enough" to x^*, converges to x^* provided that $|f'(x^*)| < 1$. Now consider the recursion,

$$x_{n+1}(m) = F(x_n(m), m) \quad , \tag{1}$$

and assume that it converges to a fixed point, $x^*(m)$, for a given m. This requires that

$$x^*(m) = F(x^*(m), m) \quad \text{and} \quad |\partial_1 F(x^*(m), m)| < 1. \tag{2}$$

Differentiating both Eqs.(1) and (2), we get the following result.

$$\begin{aligned} x^{*\prime}(m)(1 - \partial_1 F(x^*(m), m)) &= \partial_2 F(x^*(m), m) \\ x'_{n+1} &= x'_n \partial_1 F(x_n, m) + \partial_2 F(x_n, m) \quad , \end{aligned}$$

where primes denote differentiation with respect to m, and ∂_k means differentiation with respect to the k^{th} argument. The defect between x'_{n+1} and $x^{*\prime}(m)$ can therefore be estimated as follows.

$$\begin{aligned} x'_{n+1} - x^{*\prime}(m) &= (\, x'_n - x^{*\prime}(m)\,)\, \partial_1 F(x_n, m) \\ &\quad - \left[\, x^{*\prime}(m)(1 - \partial_1 F(x_n, m)) - \partial_2 F(x_n, m)\,\right] \\ &\approx (\, x'_n - x^{*\prime}(m)\,)\, \partial_1 F(x^*(m), m) \\ &\quad - \left[\, x^{*\prime}(m)(1 - \partial_1 F(x^*(m), m)) - \partial_2 F(x^*(m), m)\,\right] \\ &= (\, x'_n - x^{*\prime}(m)\,)\, \partial_1 F(x^*(m), m) \end{aligned}$$

As long as Eq.(2) is satisfied, i.e., as long as the *original sequence converges to a fixed point*, the defect decreases and $\lim_{n\rightarrow\infty} x'_n = x^{*\prime}(m)$. Higher derivatives work as well. The important thing is to recognize that simultaneously,

$$\begin{aligned} &x^{*[k]}(m)(1 - \partial_1 F(x^*(m), m)) \\ &\qquad = \Lambda(x^{*[k-1]}(m), x^{*[k-2]}(m), \ldots, x^*(m), m) \quad , \text{ and} \\ &x^{[k]}_{n+1} = x^{[k]}_n \partial_1 F(x_n, m) + \Lambda(x^{[k-1]}_n, x^{[k-2]}_n, \ldots, x_n, m) \quad . \end{aligned}$$

The demonstration of a convergent sequence then goes through exactly as with the first derivative. The *only*

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

[1] I confess to being confused about the distinction between "recursive" and "iterative."

[2] A `DA` variable carries information about derivatives of functions as well the value of the functions. It is a computer implementation of a "jet" structure [8].

0-7803-1203-1/93$03.00 © 1993 IEEE

condition that enters into play is Eq.(2), which is nothing more than the original requirement of convergence. These arguments still go through for a dimension greater than one, but the condition Eq.(2) becomes a statement about the spectral radius of the Jacobian, $\underline{\underline{\partial F}}(\underline{x}^*(\underline{m}), \underline{m})$.

III. Implicit functions

Consider the function $x(m)$ defined implicitly by the equation

$$x(m) = \cos(\, m \cdot x(m)\,) \quad . \tag{3}$$

Simple recursion can be used to construct $x(m)$ for m in the approximate range, $m \in (-1.2, 1.2)$, determined by the condition $|m \sin(\, m \cdot x(m)\,)| < 1$. A fragment of source code that uses the MXYZPTLK `DA` object (class) to implement this is shown below.

```
coord m ( 0.5 );
DA x;
x = cos( m );
for( i = 0; i < 15; i++ ) x = cos( m * x );
```

This example used a `coord` variable for m, set to evaluate derivatives of $x(m)$ at $m = 0.5$. `coord`s are the atomic `DA` variables used to start calculations, basically the implementation of a projector. The behavior of the weighted derivatives — which would be the coefficients in a power series representation of $x(m)$ — is shown in Figure 1; the first five are plotted versus loop index. Convergence is seen to be rapid, although, as suggested by the proof, a derivative does not begin to converge until the ones at lower order have already done so.

IV. Inverse functions

One of the most frequent application of recursion is to compute the inverse of a given function. For example, applying Newton's method to the equation $\tan(x(m)) = m$ provides the recursion

$$x_{n+1} = x_n - \cos x_n \,(\sin x_n - m \cos x_n) \quad ,$$

which converges to the function $x(m) = \arctan m$.[3] The recursion can be applied *directly* to `DA` variables. Using MXYZPTLK, the following short, simple C++ program follows the recursion explicitly through six steps and prints out the value and derivatives of x for a given value of m.

```
#include "mxyzptlk.rsc"
main( int argc, char** argv ) {
 const int  dim       = 1;
 const int  maxWeight = 5;
 DASetup( dim, maxWeight, dim );
 double a = atof( argv[1] );
 coord m ( a );
 DA x, s, c;
 int i, j, d[1];
 x = m;
 for( i = 0; i < 6; i++ ){
  s = sin( x );
  c = cos( x );
  x = x - c*( s - m*c );
  printf( "%-7.4lf  ", x.standardPart() );
  for( j = 1; j < 6; j++ ) {
   d[0] = j;
   printf( "%-7.4lf  ", x.derivative(d) );
  }
  printf( "\n" );
 }
}
```

When compiled and run with a command line argument of 1.2 it produced the output lines:

```
1.0198   1.0581   1.7697   4.7170   -1.6753  -30.666
0.9027   0.6776   1.9990   19.1460  119.009  531.937
0.8769   0.4280   -0.0384  7.3912   124.051  2092.93
0.8761   0.4099   -0.4015  0.5231   2.2807   101.790
0.8761   0.4098   -0.4031  0.4571   -0.3575  -0.8372
0.8761   0.4098   -0.4031  0.4571   -0.3575  -0.8414
```

Notice the repetition of the earlier pattern: derivatives settle down to their limiting value in sequence. In particular, the highest order derivatives can undergo unsettlingly large excursions before convergence kicks in. However, this is not a danger, as evaluation of higher order derivatives could be suppressed, if needed, until the lower order ones have converged.

The wonderful thing is that we could start the recursion using any `DA` variable for m, not just atomic projectors. We could, for example, use a code fragment like the following

```
coord y ( ay ), z ( az );
DA m, x, s, c;
m = sqrt( y*y + z*z );
x = m;
while( x.isChanging() ){
 s = sin( x );
 c = cos( x );
 x = x - c*( s - m*c );
}
```

to find derivatives of $\arctan\sqrt{y^2+z^2}$ evaluated at $(y,z) = (\mathrm{ay}, \mathrm{az})$. This little loop thus becomes the computational core of a `DA` -valued function that returns the arctangent of any `DA` argument.

V. A general program

Because `DA` variables possess a differentiation operation, Newton's method can be used to write a general method that works with "arbitrary" `DA` functions F. The key line that sets up the solution[4]

[3] As an acceptable seed, we could set $x_0(m) = m$ when $|m| \le 1.4$ and $x_0(m) = 1.4$ when $|m| > 1.4$.

[4] This is written inefficiently; it would be better to avoid evaluating `F` twice.

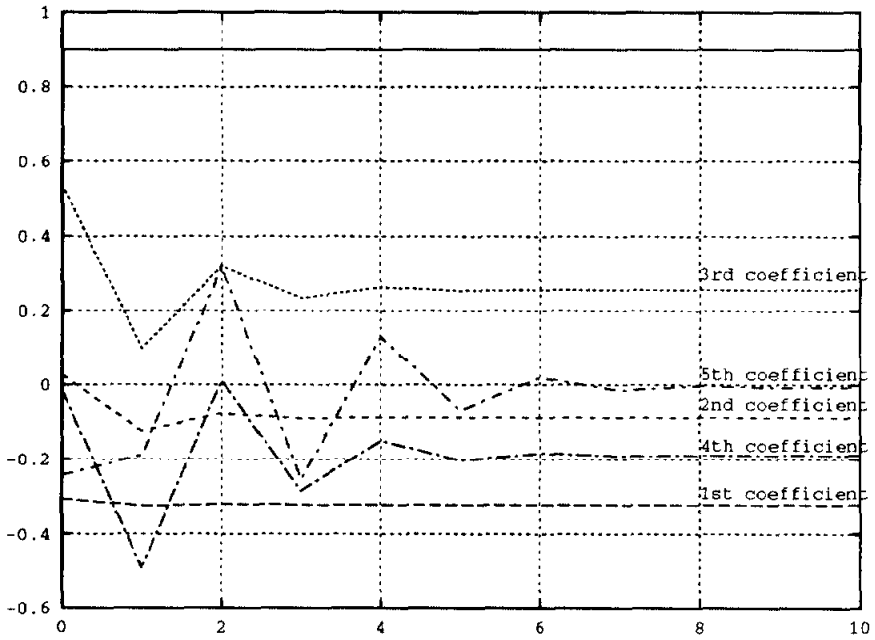

Figure 1: Behavior of the coefficients with iteration number.

```
G = x - ( F(x) / F(x).D(n) );
```

where G and x are DA variables, F is a DA -valued function of a DA argument, and .D is the differentiation operator.[5] G will be a DA variable corresponding to a single Newton step. Once it is constructed, iterating the line

```
x = G.multiEval( x );
```

will make x, with all its derivatives, converge to a zero of F. To repeat the example of Eq.(3), we then define

```
DA F( DA& x ) { return ( x - cos( m*x ) ); }
```

before entering the main function. The complete program, although short (about 60 lines) is too long to be included here. For those would like to experiment with the program and who have a C++ compiler, it and the MXYZPTLK package can be obtained *as is* via anonymous ftp from calvin.fnal.gov in the directory /pub/outgoing/michelotti/mxyzptlk or /pub/outgoing/michelotti/beamline.[6]

References

[1] Martin Berz. Differential algebraic description of beam dynamics to very high orders. *Particle Accelerators*, 24(2):109, March 1989.

[2] G. Corliss and A. Griewank, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, 1991. Philadelphia, PA.

[3] 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*, 24:91, 1989.

[4] Jean Charles Gilbert. Automatic differentiation and iterative processes. *Optimization: Methods and Software*, 1(1):13–21, 1992.

[5] Leo Michelotti. MXYZPTLK and BEAMLINE: C++ objects for beam physics. In *Advanced Beam Dynamics Workshop on Effects of Errors in Accelerators, their Diagnosis and Correction. (Corpus Christi, Texas. October 3-8, 1991)*. American Institute of Physics, 1992. Conference Proceedings No.255.

[6] ——— MXYZPTLK: A practical, user-friendly C++ implementation of differential algebra: User's guide. Fermi Note FN-535, Fermilab, January 31, 1990.

[7] ——— A note on the automated differentiation of implicit functions. Technical Memo 1742, Fermilab, June, 1991.

[8] Gordon Pusch. Private communication.

[9] Joseph Fels Ritt. *Differential Algebra*. American Mathematical Society, New York, 1950.

[5] n is an integer array needed by .D; essentially, n tells .D which derivative is desired.

[6] These files may be moved eventually.