EFFICIENT C++ LIBRARY FOR DIFFERENTIAL ALGEBRA

John R. Cary, Tech-X Corporation and University of Colorado, Boulder, USA
S. G. Shasharina, Tech-X Corporation, Boulder, USA

Abstract

Differential Algebra is heavily used in accelerator physics for rapid integration, long-term stability studies, and non-linear map analysis. C++ with operator overloading is a natural language for implementing DA, but sometimes is too slow. We created a prototype of a DA vector class, which, due to use of features of multiplication tables, expression templates and reference counting is faster than other C++ packages.

1 INTRODUCTION

Numerical Differential Algebra methods have found increasing use as they can be applied to a wide variety of systems. In the study of system sensitivities or optimisation, numerical differential algebra methods allow one to determine how a system varies with parameters to machine accuracy with little additional programming effort. There are also applications to dynamical systems, such as accelerators. DA use has been described in the literature (see Refs. [1-2]) and comes from the fact that one can obtain non-linear map by integrating high-order DA vectors in the differential equations describing particle motion.

In our definition (more narrow than general differential algebras), Differential Algebra of order n is formed by vectors, whose components are obtained by n-order Taylor expansion of functions in d-dimensional space. To explain this, we must describe the representation of a DA vector in the class. In order to represent all functions in terms of DA vectors, we need to define basic arithmetic operations (like +, *etc.) between DA vectors and implement them numerically. There are several numerical libraries implementing this, but some of them are written in computationally archaic languages (see Refs. [3–4]), others are written in C++, but relatively slow for reasons we describe below (Refs. [5-7]). We created a prototype of a new C++ library (TXDA) which overcomes usual limitations of C++ and exploits some features of multiplication tables for fast multiplication.

2 OVERCOMING TRADITIONAL C++ LIMITATIONS

C++ has many advantages: it is object oriented, has operator overloading and is widely used. But its advantages (abstraction) lead to substantial penalties in speed, if operators are implemented straightforwardly as described in most textbooks.

2.1 Expression Templates

Simple-minded operator overloading leads to creating many extra temporaries and extra loops. Lets consider an example when 3 vectors are added: a = x + y + z. First, a temporary, temp, for holding (y + z) is created and the first loop for assigning the value is performed. Then second temporary, temp2, for (x + temp1) is created and looping is carried out again. Finally, last loop assigns a to temp2. Expression Template (ET) technique (Ref. [8]) allows one to effectively unroll these loops into one, so that the equivalent code will be:

```
for(int i = 0; i<length; ++i) {
        *a = *x + *y * +*z;
         a++; x++, y++, z++;
        }
```

Some people will say that this is just a hand-coded C, so what is the point? The point is that ET allows to keep elegant abstractions of C++ (so that you can use overloading and write a = b + c + d for such objects as matrices, vectors etc.), but the computational work will be equivalent to hand-coded C. This technique was successfully implemented in our DA vector class for addition, subtraction, multiplication and division by a scalar. Resulting speed of these operations is order of magnitude faster than in ZLIB and LEGO. Examples of the comparisons results are shown on Figs. 1 and 2: our DA vector class is an order of magnitude faster than codes using traditional operator overloading.

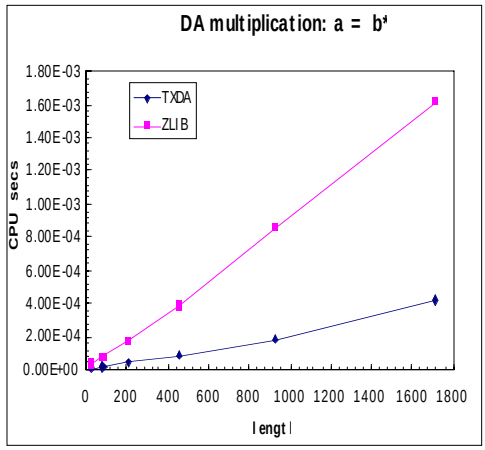

Figure 1: Comparison of the execution time for by two double numbers in ZLIB and TXDA (DAV) versus vector length. The results are obtained on SGI/Indy using the KCC compiler

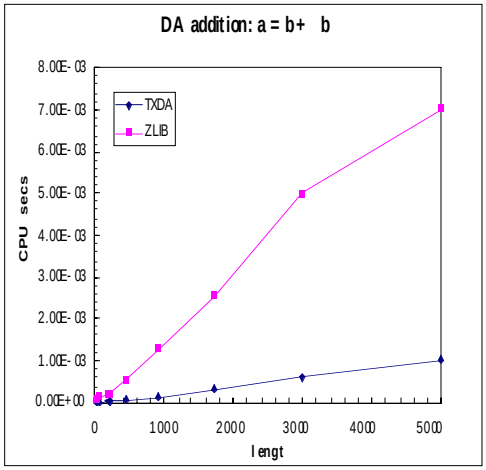

Figure. 2: Comparison of the execution time of addition in ZLIB and TXDA (DAV). The results are obtained on an SGI/Indy using the KCC compiler.

2.2 Reference Counting

An object is copied each time it is passed by value into a function, each time it is returned by value, and each time it is assigned. Since DA vectors and DA maps can easily be large (represented by thousands of real numbers), such copies can take a significant amount of time and memory even when they are not necessary. For example, when returning a vector by value, the object is copied to a variable in the external (to the method) namespace, when all that is needed is to transfer the object from one namespace to another.

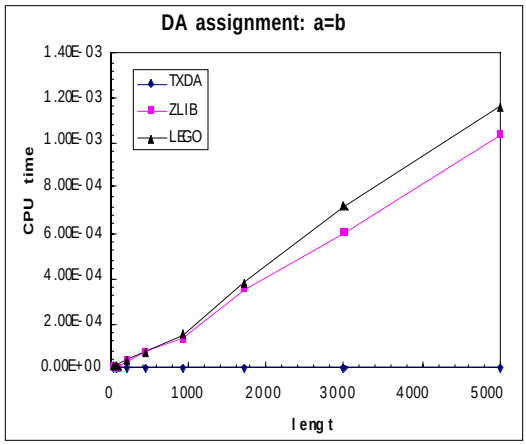

Figure 3: Comparison of the execution time of assignment in ZLIB and TXDA (DAV) versus vector length. The results are obtained on SGI/Indy using the KCC compiler.

The reference counting method allows "shallow copying" (through pointers) with security of the regular copying. It puts the representation of the object in an interior *letter* object and adds to the letter class a counter that keeps track of the number of references there are to it. At copy time, one simply increments the number of references to the letter object. Only when an object is changed is a new copy of the letter object made. Upon destruction, the counter is decremented, and the letter object is destroyed only if there are no more references to it. We implemented reference counting in TXDA and results of the speed comparison are shown on Fig. 3. Our results lie on the axis (execution time < 1.e-11) and do not depend on the vector length.

3 MULTIPLICATION

It is natural to use a multiplication table for implementation of multiplication of DA vectors. After studying literature we concluded that using lexicographically graded multiplication tables, as described by Alex Dragt (Ref.[9]) would be advantageous. An ungraded multiplication table is obtained by looping through the index of the first factor (external loop) with the internal loop running through the index of the second factor. For each pair of indices there is an integer that corresponds to the product index to which this coefficient product contributes. If there is no such index (because the product exceeds the order of the DA) the row is left out of the table.

Table 1: Multiplication table: product index (ip) for given indices (if1, if2) of the factors.

ip	if1	if2
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5
6	0	6
7	0	7
8	0	8
9	0	9
10	0	10
11	0	11
12	0	12
13	0	13
14	0	14
15	0	15
16	0	16
17	0	17
18	0	18
19	0	19
8	2	3
11	2	4
13	2	5
14	2	6

16	2	7
17	2	8
18	2	9
12	3	4
14	3	5
15	3	6
17	3	7
18	3	8
19	3	9
0	0	0
4	1	1
7	2	2
9	3	3

Table 1. Multiplication table giving product index (ip) for given indices (if1, if2) of the factors.

Table 1 shows a part of the table for three-dimensional, order-3 DA vectors, where ip is the index of the product, if1 is the index of the first factor, and if2 is the index of the second factor. In the above table we have shown only the entries with $if1 \leq if2$. We found that we obtained substantially better performance by breaking our table up into the subgroupings shown above. For the asymmetric parts of the table, $if1 \neq if2$, we used only the above loops and added in the other terms with the same lookup, i.e., c[ip] += a[if1]*b[if2]+a[if2]*b[if1]. Then we add the symmetric parts. For each of these subtables one is simply stepping through the second index, and the first index is held constant. Hence, the only lookup is that of the product coefficient index. Indeed, for parts of the table, where if1=0, even the product index need only be incremented, not lookup up each time. Ultimately we implemented this in very efficient pointer arithmetic. This was implemented and gave us an increase in speed of a factor of 2-8 over existing C++ libraries (see Fig. 4). For tests we used dimension 6 and order up to 12.

2 CONCLUSION

We created a prototype of DA library which combines benefits of object oriented approach (written in C++) with the efficiency of procedural languages.

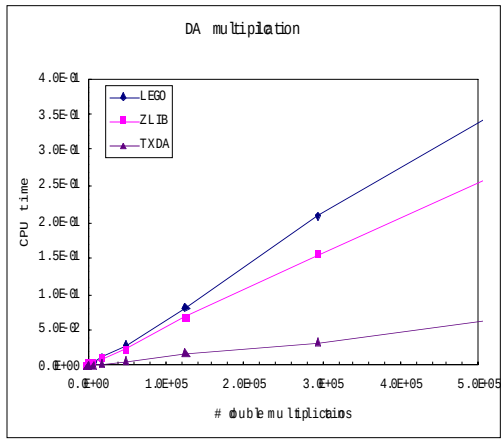

Figure 4: DA multiplication time for DA versus the number of primitive multiplications of real numbers.

REFERENCES

[1] M. Berz, "Differential algebraic Description of Beam Dynamics to very High Order," Particle Accelerators, **24**, 109 (1989).

[2] M. Berz, "Differential algebraic Approach of Normal Form Theory," Inst. Phys. Conf. Ser. No 131 (paper presented at Int. Workshop Nonlinear Problems in Accelerator Phys. Berlin, 1992.

[3] M. Berz, COSY INFINITY version 7 Manual, MSUCL-977 (1996), Michigan State University, East Lansing, MI 48824; http://www.beamtheory.nscl.edu/cosy/.

[4] A. Dragt et. al , "MaryLie 3.0 User's Manual," University of Maryland Physics Department, 1994.

[5] L. Michelotti, "MXYZPTLK and BEAMLINE: C++ Objects for Beam Physics," AIP Conf. Proc. No. 255 (Proc. Advanced Beam Dynamics Workshop on Effects of Errors in Accelerators, their Diagnosis and Correction, Corpus Christi, Texas, 1992).

[6] N. Malitsky, A. Reshetov and Y. Yan, "ZLIB++: Object Oriented Numerical Library for differential algebra," preprint SSCL-659, 1994.

[7] Y. Cai, M. Donald, J. Irwin, Y. Yan, "Lego: A Modular Accelerator Designer Code,"SLAC-7642, August 1997.

[8] T. Veldhuizen, "Expression Templates," C++ Report, v. 7 No. 5 June 1995, pp. 26-31.

{9] A. Dragt, M Venturini, "Design of Optimal Truncated Power Series Algebra. Routines: II. Computing Sums and Ordinary and Lie Products of Polynomials Usong Monomial Indexing or Linked Lists," University of Maryland, September 1996 (Draft).