FINAL TECHNICAL REPORT

Title: Efficient and User Friendly C++ Library for Differential Algebra

Grant No. DE-FG03-97ER82499/A000

PI = Svetlana G. Shasharina, Business Official = John R. Cary

Signature: _____

A handwritten signature in black ink, consisting of a large, stylized 'S' followed by a horizontal line and a small flourish.

Title: P.I.

Date: September 29, 1998

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, make any warranty, express 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.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

1 Phase I objectives

In Phase I we proposed the following tasks:

- Task 1: Identify the units of the DA library, i.e. determine the abstract data types of the DA and the relations between them. Develop the interfaces (.h files) for the DA vectors.
- Task 2: Implement the DA vector class with garbage collection and expression templates for optimizing all overloaded operators by minimizing creation of temporaries and fusing loops.
- Task 3: Implement the prototype GUI for instantiating systems from files and invoking the DA methods.
- Task 4. Develop a suite of tests for the DA vector class and the needed utilities classes.
- Task 5. Write the final report on this work. This will include documentation on the use of the code.

We have completed these tasks. In this section we discuss the results of our work.

2 Completion of Task 1

The task was to identify the units of the DA library, i.e. determine the abstract data types of the DA and the relations between them and develop the interfaces for the DA vectors. Our analysis of existing DA libraries and future needs shows that a DA library should contain the following components.

1. DA vector and map classes with reference counting and expression templates and optimized multiplication. This includes all arithmetic operators and special functions and norm.
2. Library for DA maps including
 - a. Map inversion
 - b. Lie factorization
 - c. Calculation of normal forms
 - d. Symplectification (Cremona factorization)
3. Polymorphic integrators for integrating of ODE's using DA vectors, real numbers, complex numbers, and other objects for which the arithmetic operations are defined.

The interface of the DA vector is determined by the need to be able to set and get individual components, read and write these objects, efficiently carry out the arithmetic operations among DA vectors or with scalars, and integrate ordinary differential equations involving these objects. In addition, we would like our vectors to have coefficients that can be of

any arithmetic type: real numbers, complex numbers, even other DA vectors. The latter is particularly important for normal form theory, especially for dissipative systems. We also need to be able to carry out run time conversions between DA vectors of different orders and over different types.

A design that satisfies all of these conditions is to have DA vectors, which we denote TxDAV (the V representing vector, as they form a vector space), templated over type, order, and dimension:

```
template <class T, int dimension, int order > class TxDAV : public class
    DABase <dimension> {

private:
    // Data representation

public:
    // Constructors, including copy constructors, and destructor
    // Assignment operator
    // Accessors (setters and getters)
    // Arithmetic operations: with self, postarithmetic with scalars
    // Derivatives and integrals
    // Evaluation for a given deviation
    // Boolean operations (=, !=)
    // I/O: (<<, >>)
    // friend functions and operators: elementary functions, abs, sqrt,
    // prearithmetic (preaddition etc.) by scalar
};
```

The base class, DABase, contains index tables and multiplication tables. We will discuss this shortly. This design allows us to convert automatically between types and orders as appropriate. For example, a copy constructor from TxDAV<double> to TxDAV<complex> will automatically make the correct conversions if the member templated copy constructor,

```
template<class T2> TxDAV(const TxDAV<T2,order,dimen>&);
```

simply does a term by term copy. Similarly, having the copy constructors,

```
TxDAV (const TxDAV<Type, dimension-1, order>&);
TxDAV (const TxDAV<Type, dimension+1, order>&);
TxDAV (const TxDAV<Type, dimension, order-1>&);
TxDAV (const TxDAV<Type, dimension, order+1>&);
```

will allow runtime conversion between different orders and dimensions, as the C++ compiler recursively applies such constructors. For speed one can add copy constructors of greater differences rather than relying on recursion. We note that for advanced features such as template parameter recursion and member template functions one must have a fully ANSI compliant

compiler.

The friend operators of this interface are needed for arithmetic operations of the form, $2+x$ or $2*x$, where x is a DA vector. This is because the rules of C++ state that the operator belongs to the first operand. In addition, one would like functions, e.g., `double abs(x)` and `TxDAV sin(const TxDAV& x)`, where x is a DA vector, to be able to take advantage of compile time polymorphism. These functions must know the internal structure of the DA vector and, so, must be friend functions.

Templating over order and dimension has several advantages. The first is that compilers can make aggressive compile-time optimizations of loop when the limits are known at compile time. The second is that one can specify special operators for particular instances. This is useful, e.g., for multiply `operator*`, which can be coded much more efficiently for the low orders and low dimensionality. Thus, `operator*(const TxDAV<Type, dimension, 1>&)` has only a single loop of length equal to the dimension of the DA vector.

We now discuss the base class `DABase`, which contains index tables and the information needed to multiply two DA vectors. To explain this, we must describe the representation of a DA vector in the class. The coefficients $a_{i_1, \dots, i_N}$ for an N -dimensional DA vector (or truncated power series),

$$a = \sum_{i_1 + \dots + i_N < M} a_{i_1, \dots, i_N} \delta_1^{i_1} \dots \delta_N^{i_N}, \quad (1)$$

of order M , are placed into a one-dimensional array representation in or objects. This is known as the indexing scheme.

Many indexing schemes are useful. Regardless of the scheme chosen one must create a correspondence between the exponents, $(i_1, \dots, i_N)$ and the location i of that coefficient in the linear array. This is given by some function $i(i_1, \dots, i_N)$. To invert this, one must have the exponents as a function of the index, $i_k(i)$. For a given order and dimension, these quantities are fixed for all objects. Indeed, for graded lexicographic schemes, which we will use, one need make only one table for the maximum order, and the tables for lower orders are included within it. Hence, the table including base class is templated over dimension only.

```
template <int dimension> class DABase {
public:
    DABase();          // Default constructor - called at creation of a TxDAV
    DABase(int);       // Called at link time with maximum order to be created
```

```

    int Index(const int* exponents);          // Get index from exponents
    const int* Exponents(int);               // Get exponents from index
};

```

We create a single global instance of this class for each dimensionality at link time. The statement

```
DABase<6> database6(12);
```

creates the static index and multiplication table up through 12th order. One must have a similar statement for each dimension.

To multiply two DA vectors, $c = ab$ one must add to the coefficient of index k of the result all of the products of the appropriate coefficients of the two factors,

$$c_k = \sum_{k_i=i_i+j_i, \dots} a_i b_j. \quad (2)$$

One way to represent this is as a table $k(i,j)$ of product indices as a function of factor indices. However, we have found a simpler representation, where the table is separated first into two tables, one for the symmetric terms and one for the asymmetric terms of the sum (2). The asymmetric table is then separated further into multiple tables in such a way that cache utilization is optimal, as discussed later in the proposal. These multiplication tables are also included in our class DABase and created only once, at link time.

3 Completion of Task 2

The task was to implement the DA vector class with reference counting and expression templates for optimizing all overloaded operators by minimizing creation of temporaries and fusing loops. In Phase I we fully implemented primary operations of the DA vector class with the data of the type double. These include assignment, copy constructor, addition, multiplication between DA vectors, and multiplication by a number. In fact we did more in this area than proposed. We also optimized multiplication by elimination of parasite operations, and through a literature examination we determined the existence of the blocking method, which should further optimize multiplications. In this section we show the results of our optimized implementation of reference counting, expression templates, and multiplication.

Fig. 1. 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.

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. The reference counting method 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

Fig. 2. 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.

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.

In Fig. 1 we show a comparison of speed of assignment for DA vectors of our library (class `TxDAV`), with that for vector from the ZLIB libraries (LEGO was typically slower than ZLIB for most operations). The graph shows that copying using reference counting is constant and fast, with a copy time less than that for the regular copies of other libraries for first-order DA vectors of small dimension.

Expression templating has been successfully applied to optimize binary operations that do not mix indices of DA vectors in the right-hand side. The key to this technique is ability to assign the object (DA vector) to an expression, so that the evaluation of the i^{th} component of the result is postponed till the moment when the whole expression is parsed. In contrast, the regular practice in C++ is to have assignments only to the objects of the same class. This leads to pairwise calculation of expressions, so that for example in expression $v1 = v2 + v3 + v4$, the compiler first creates a temporary out of $\text{temp1} = (v3 + v4)$ calculates it with the first loop, then creates $\text{temp2} = v2 + \text{temp1}$, calculates it with second loop, after that assigns temp2 to $v1$. With the expression template technique two loops and the creation of one temporary are eliminated.

Figure 2 shows the results of implementing the expression template technique for

multiplication by a scalar twice. As seen in the figure, the CPU time still scales with the vector length, but more slowly by a factor of five or so. Figure 3 compares the expression template technique with the usual methods for the addition of three DA vectors. Again the CPU time is seen to increase more slowly with length.

Fig. 3. Comparison of the execution time of addition in ZLIB and TXDA (DAV). The results are obtained on an SGI/Indy using the KCC compiler.

Multiplication is the most time consuming and complicated operation over DA vectors. In the subsection devoted to the structure of the TxDAV class, we briefly described multiplication tables. We have discovered many features that allowed us to speed up multiplication compared to ZLIB and LEGO. We partly implemented the results of our studies (more on this approach can be found in description of our objectives) and obtained very promising results: an increase of speed from 2 to 8 times compared to existing C++ classes depending on the compiler.

Figure 4 compares our timing results for multiplication with those of existing libraries. This figure shows that we have obtained a factor of approximately eight in increase of speed. To put this in perspective, this is a software increase of power equivalent to changing from a 120 MHz Pentium to a 500 MHz DEC alpha chip.

In summary, we have implemented all primary operators of the DA vector classes. Addition, multiplication by number, copying and multiplication have proven to be faster than any

of existing C++ codes by factors of 2 to 8.

Fig. 4. DA multiplication time for relatively large DA vectors (6 dimensions, order < 12) versus the number of multiplications of real numbers for DA vectors of various sizes. The results are obtained on an SGI/Indy using the KCC compiler.

4 Completion of Task 3

Fig. 5. A prototype of DA calculator: vectors are read from files chosen through the menu File. The result is put into a new file.

The task was to implement the prototype GUI for instantiating systems from files and invoking the DA methods. A screen shot of our implementation of this GUI is shown in Fig. 5.

From the file menu one can open a files containing a DA vectors and use them as either operand 1 or operand 2. One can also specify a file into which the result will be place. One can choose any of the arithmetic operators by pressing the appropriate button. Finally, pressing the calculation button causes the result to be calculated and placed into the result file.

5 Completion of Task 4

The task was to develop a suite of tests for the DA vector class and the needed utilities classes. To complete this task we built an executable that links in our library along with the LEGO and ZLIB libraries. Our bench code uses the primary arithmetic operators for each of the DA libraries and compares the results term by term. It runs and was tested on an SGI/Indy with the KCC compiler and a Gateway-2000/Pentium-Pro 200MHz with g++ compiler. All answers agreed.

6 Completion of Task 5

The task was to write the final report on this work and include documentation on the use of the code. This report accomplishes this task.

7 Summary

We fully completed all the tasks of Phase I, thereby demonstrating the feasibility of developing a robust, object oriented library for differential algebra that will be as fast as any coded in a procedural language.