EFFICIENT DIFFERENTIAL ALGEBRA COMPUTATIONS*

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

Abstract

Numerical Differential Algebra (DA) is a powerful tool for studying non-linear motion in accelerators, beam and optics devices. Implementation of DA is the easiest in an object oriented programming language, especially C++. In addition to standard object oriented features, C++ allows for operator overloading and static polymorphism via templates. In this paper we discuss (1) use of templates for polymorphic use of the code, so it can treat both dynamical variables and DA maps, (2) methods of optimisation for speed needed for efficient use DA in accelerator physics and other applications.

1 DIFFERENTIAL ALGEBRA IN LAYMAN'S TERMS

Dynamical systems – accelerators, beam and optics devices – can be represented by a mapping of initial conditions (depending on parameters a) to final values variables $z_f = M(z_i, a)$. Calculation of this map constitutes a very complicated task, since the motion of the particles is generally highly non-linear. The fields and parameters of the system are usually known to a certain order in a Taylor expansion in a deviation from some reference in the dynamical space. That is why it makes sense to limit the accuracy of map calculations to the same order and treat all functions of calculation as truncated Taylor series. First order derivatives of the map are known as transfer matrix. Nonzero higher order derivatives of the map represent aberrations, while derivatives with respect to parameters are sensitivities.

Coefficients of the truncated expansion can be used as DA representation of functions and, in fact, constitute a vector space. One can show that this space is a differential algebra so that it has arithmetic operations, inversion, roots and derivation [1]. DA representations of components of identity function, z_i (i=0,…,d, with d being the dimension) form algebra generators, vectors with all but one component equal zero. The nonzero component, 1, corresponds to first order derivative with respect to z_i. For example, the zeroth generator will have components {0,1,0,….,0} (with the first position reserved for the constant term). Note, that the order of coefficients in vector representation is a matter of choice, although lexicographical order is most popular (see [2]).

The whole realm of DA applications is very rich and powerful. The most commonly used application is calculation of non-linear maps for solving equations of motion in a range of initial conditions. Since the map is a solution of equations of motion and is equal identity at t=0, it can be found by propagating of identity map through the system. Identity map can be considered a d-dimensional vector of generators. Hence is the recipe for generating maps: one has to replace all functions in the integration algorithm with their DA representations and propagate the identity map. The resulting "matrix" will give the system map to the desired order. Once the map is found, there is no need to integrate trajectories with different conditions: one can calculate the final state by substituting monomials corresponding to initial condition into the map. This procedure is much faster than integration, so it can be used for multiparticle calculations and long-term stability studies.

2 C++TEMPLATES AND GENERATION OF MAPS

In this section we show how the recipe for generating maps discussed above can be elegantly implemented in C++. First we need to discuss C++ templates mechanism. Then we give an example of its application to generating a simple map.

2.1 C++ Templates and Static Polymorphism

Templates is a powerful tool for providing polymorphism used extensively in generic programming (see [3]). A typical example of a template function is `Swap()` function performing sequence of actions not dependent on type of the arguments:

```
template <class T> function Swap(T& a, T& b) {
  T temp(a);
  a = b;
  b = temp;
}
```

The type of the template parameters is passed to compiler at compile time when compiler substitutes the actual type into the templated code (template instantiation) and uses templated code as if it is not templated but indeed has the needed type inserted. This polymorphism is called static polymorphism since the choice of the type happens at compile, not run time. Thus, the `Swap()` function become

* Work supported in part by DOE/SBIR Grant No. DE-FG03-97ER82499

"untemplated" as soon as some concrete type is used for its arguments:

```
int main() {
  SomeType x (2);
  SomeType y(3);
  Swap(x, y);
}
```

If class `SomeType` declares copy constructor and assignment operator private, the main program will not compile. This gives a typical example of template programming: most of templates are built with some assumptions (template constraints) for the template parameters. Note that C++ allows developers to create free template functions, template classes and template member functions in non-template classes.

2.2 Templates, Traits and Map Generation

Now we are ready to explain how we can generate maps, provided that we know how to integrate usual dynamic variables. Imagine that we have class `System` which has a template function `Advance()` for calculation of final value the vector describing the dynamical variables in 6-dimensional space. For simplicity, we will use a function propagating through a simple drift. The argument will be encapsulated in a vector container whose choice depends on a user:

```
void template <class VecType>
System::Advance(VecType& z) {
  double pz  = CalcPz(p);
  double lenOverPz = length/pz;
    p[4]+=lenOverPz*
     (InvReferenceBeta()+p[5]) +
     length*(1+Eta()*Delta_S)*
     InvReferenceBeta();
    p[2]+=p[3]*lenOverPz;
    p[0]+=p[1]*lenOverPz;
  }
```

`VecType` can be any of multiple containers for double vectors (`std::vector<double>`, `valaray<double>` etc.). Now we want to generalise this function so that it can treat not only double vectors, but also maps.

To make the receipe described in the previous section work, we have to overload `operator[](int)` in `DAMap` class so that it returns a corresponding DA vector. In addition, we assume that DA maps and vectors are implemented as template classes with type of coefficients being the template parameters, so that we can describe real or complex DA's used for normal form analysis. We also found advantageous to have dimension and order as template parameters, but for simplicity will omit them here. Thus, schematically the `DAMap` class will have the following structure:

```
template <class U> class DAMap {
      DAVector<U>* data;
    public:
      const DAVector<U>& operator[](int i) const
        {return data[i];}
       DAVector<U>& operator[](int i)
        {return data[I];}
    //etc.
};
```

Now, lets go back to generalisation of the `Advance()` function. Instead of `VecType` we now imagine more general template parameter which can include not only vectors, but also maps, since we made sure that `DAMap::operator[](int)` is defined. The first question coming to mind is what should we put instead of type declaration "`double`"? Evidently, this type should be the same as a type of variable returned by `operator [](int)`. In case of a double vector, it should be `double`, in case of a DA map, this should be a DA vector! If we want to use one generalised `Advance()` function for both dynamic vectors and DA maps, how do we code this information so it works for both?

The solution is in using traits (see [4]). Traits are special classes, which provide mechanism to associate certain functions, values or types with particular classes and access them in a uniform way. For example, we often need to know what is the type of data is contained by the vector? In case of `std::vector<T>` it will be `T`. In case of `DAMap<U>`, it should be a `DAVector<U>`, since any map is just a vector of DA vectors. Similarly, we might want to know a size of the vector, which will be function `size()` for `std::vector<T>` and `DAMap<U>::Dimension()` for DA maps. This information can be encapsulated in the following traits classes. First we define a general template class which, being unspecialised, does not have anything:

```
template <class U> struct UnaryTraits {};
```

Now we can partially specialise it for vectors and maps:

```
template <class U>
UnaryTraits <std::vector<U> > {
//Define the type of data
  typedef U ValueType;

//Get size of the vector
  static int GetSize(const std::vector<U>& u) {
     return u.size()}
};

template <class U>
  UnaryTraits <DAMap<U> > {
  //Define the type of data
    typedef DAVector<U> ValueType;

  //Get size of the vector
    static int GetSize(const DAMap<U>& u) {
      return u.Dimension();}
};
```

With traits defined, we can rewrite our `Advance()` function in such a way that it will treat both vectors of

dynamic variables and DA maps by accessing the correct `ValueType`:

```
void template <class AnyType>
System::Advance(VecType& z) {
  typedef typename
    UnaryTraits<AnyType>::ValueType ValueType;
  ValueType pz  = CalcPz(p);
  ValueType lenOverPz = length/pz;
    p[4]+=lenOverPz*
     (InvReferenceBeta()+p[5]) +
     length*(1+Eta()*Delta_S)*
     InvReferenceBeta();
    p[2]+=p[3]*lenOverPz;
    p[0]+=p[1]*lenOverPz;
  }
```

Note that function `CalcPz()` should be rewritten and become a template function to return the appropriate value:

```
template<class T> T System::CalcPz(T& p) {
   T result;
   //calculate result
   return result;
  }
```

In the main program, one has to decide which quantity to propagate and use the `Advance()` function:

```
  int main() {
//Create the system:
  System sys();
//Set system's parameters (not shown)
//Create a 6-dimensional double:
    std::vector<double> x(6);
//Set x to some intial conditions(not shown)
//Propagate x through the system
    sys.Advance(x);

//Create a default 6-dimensional map
//of the 4 order
    DAMap<double> map(6,4);
//Set this map to identity (not shown)
//Find the map of the system:
  sys.Advance(map);

//Vector x now has been propagated once,
//now do it the second time using the map
//(syntax arbitrary):
  x.Map(map);
//etc.
  return 0;
}
```

Thus, use of template functions and trait mechanism allows for true reuse of the code, so that the same function is used to integrate dynamic variables and generate non-linear maps.

3 OPTIMIZATION FOR NUMERICAL EFFICIENCY

3.1 C++ and Performance

As many scientist like to complain, C++ can be inefficient for numerical for number crunching. There are many reasons for that, one of which is that it is very easy to write a bad C++ code. Another, objective, reason, sometimes called a burden of abstraction, is that ordinarily overloaded operators are defined in such a way that they create a temporary out of binary operation whenever more operations are present in the expression. For example, adding 3 vectors together, a = x + y + z; will create temp1 = (y + z) with one looping over indices performed. Then it will calculate temp2 = x + temp and assign a to temp2. One can easily see that we can get rid of at least one temporary and one loop if we just loop through all 3 vectors together. In our previous work (see [5]), we have shown how to overcome this burden: by using of expression templates (ET). ET makes addition/subtraction and multiplication/division by a scalar as fast as hand-coded C and speeds up these operations by an order of magnitude!

Another common technique, reference counting, allows safe copying via pointers, which saves enormous time, whenever assignment or copying in and out of functions is performed (see [5]).

Unfortunately, these methods do not contribute much to efficiency of multiplication of DA vectors. Typically, DA calculations deal with 6-dimensional maps in 1 to 12 orders. This implies very long vectors and situations when to obtain a product of 2 vectors, compiler has to perform half-a-million elementary multiplications between double numbers is not uncommon. This poses a task of speeding up multiplication and operations using it (powers). Note that solution to this problem almost does not depend on the choice of programming language.

We discovered several techniques useful for optimisation of multiplication. One is optimisation of multiplication tables, as we described in [5]. For the sake of completeness, we will remind how it is done in the next section. Then we will comment on effects of programming style on efficiency.

3.2 Optimisation of Multiplication Tables

Once the order of monomials in DA vectors is set, one can obtain the multiplication table 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 there is an integer that corresponds to the index of the product to which this pair contributes. For example, lets multiply two second order 2-dimensional DA vectors:

c = a*b;

a=a0+a1*x+a2*y+a3*x*x+a4*x*y+a5*y*y=

{a0, a1, a2, a3, a4, a5}};
b=b0+b1*x+b2*y+b3*x*x+b4*x*y+b5*y*y=
{b0, b1, b2, b3, b4, b5};
c ={c0,c1,c2, c3, c4, c5}.

The multiplication table can be easily found by multiplication of polynomials):

ip	if1	if2
0	0	0
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5
1	1	0
3	1	1
4	1	2
2	2	0
4	2	1
5	2	2
3	3	0
4	4	0
5	5	0

To obtain the product c with this table, one has to go through the table and perform 3 lookups for each step:

```
for(int i=0; i<table.length(); ++i)
    c[table[i][ip]]+=a[table[i][if1]*b[table[i][if2];
```

Such naïve approach leads to pretty slow multiplication. One can improve the speed by reorganising the table in the following way. First, let us split it into symmetric and asymmetric parts. Symmetric table has if1 = if2 with these indices being simply incremented in the table, while the product index is still to be looked up. Then, in the asymmetric part, we leave only lines where the first factor is smaller than the second (this leads to getting almost a factor of 2 of speedup). For our example, we obtain the following symmetric table:

ip	if1	if2
3	1	1
5	2	2

and 2 asymmetric tables:

ip	if1	if2
1	0	1
2	0	2
3	0	3
4	0	4
5	0	5

ip	if1	if2
4	1	2

One can see that to obtain the product, one has to loop through the symmetric table adding a[if1](b[if1], where if1 does not have to be looked up. Then one goes through all asymmetric tables, adding a[if1]*b[if2]+a[if2]*b[if1]. The number of lookups can be reduced even more, if one notices that each asymmetric table has the first index equal to the number of the table i (starting from 0), while the second index can be found by incrementing an integer initially equal to (i+1). As a result, there will be only one look up for each row of the table, instead of 3.

3.3 Programming Style

When we compared different approaches to DA implementation, we discovered that general programming style makes a big difference in performance. One should try to avoid lookups, dereferencing, if-statements. Thus, we discovered that pointer arithmetic is faster than operating with dereferenced arrays, so that the following code:

```
double* xPtr = x[0];
double* xEndPtr = x[5];
double* yPtr = y[0];
double* aPtr = a[0];
while (xPtr<xEndPtr) {
 *a = *x + *y;
  a++; x++, y++;
}
```

is faster than:

```
for(int i=0;i<a.size;++i)
  a[i] = x[i] + y[i]
```

We used pointer arithmetic in the numerical implementation of multiplication combined with optimisation of multiplication tables, as described above, to obtain pretty good results. Our multiplication is 5-8 (depending on platform and compiler) faster than other C++ codes. This is shown on Fig. 1.

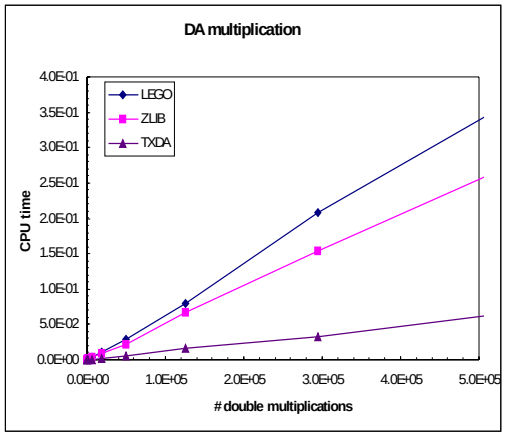

Figure 1: DA multiplication time for DA versus the number of primitive multiplications of real numbers.

4 FUTURE DIRECTIONS FOR OPTIMISATION

4.1 Optimisation for Cache

In our studies of multiplication, we noticed that CPU time taken by one multiplication of DA vectors grows linearly with the number of elementary scalar multiplications up to a point where it changes its slope (see Fig. 1). Such behaviour is a typical manifestation of cache problems.

Cache is a small fast memory holding recently accessed data, designed to speed up subsequent access to the same data. All data requested by the program is fetched into cache and stays there until it is flushed by new data. Cache operates on presumption of spatial and temporal locality: it expects that the data requested by the program will be used soon again, as well as data close to the requested. Hence, data is cached in memory lines, which are typically larger then the specifically requested data. The access for main memory is much slower than accesses to cache. Thus, when the size of the data needed to perform a multiply reaches a certain size, the calculation slows down, because of accesses to main memory rather than cache. This fact was noticed by computer scientists optimising multiplication of matrices. A special technique "blocking" (sometimes also called "tiling") has been developed (see [6]). It divides matrices into blocks with sizes optimised for particular cache size, so that the operations are able to reuse data staying in cache. We can not directly take this idea since we deal with the multiplication table and different objects. But we will reorganise the multiplication tables into subtables, so that the indices of each subtable stay close to each other within some blocking size. The optimal size will depend on the platform. The idea to efficiently utilise the memory hierarchy is natural, because while the speed of processors has been increased rapidly, it has not accompanied by a similar increase in the memory speed.

4.2 Optimisation of low-order DA's

In the case when the order of the DA is low (1 or 2), the multiplication table looks particularly simple. We will provide a special version of the library for such cases, which will be used in automatic differentiation. Templating over order permits the definition of special cases.

4.3 Optimisation of powers

These operations will not be implemented in terms of multiplication operator, but rather on a lower level (similar to multiplication), because explicit use of the symmetry of the factors in the product will be advantageous for speed. Thus, instead of x^8 = x*x*x*x*x*x*x*x one should make compiler do $(x^4)*(x^4)$.

5 CONCLUSIONS

We have developed a prototype of C++ library for Differential Algebra, which has a potential to become the most efficient C++ library. It has expression templates for efficient addition/subtraction and multiplication/division by a scalar. It has reference counting for safe and rapid copying, whose speed does not depend on the length of vectors. Implementation of multiplication uses optimised multiplication tables and is free of parasite operations. The resulting speed of our classes is 5-8 times faster than other C++ libraries. The classes are polymorphic, so that we can have DAVector<double>, DAVector<complex> for normal form analysis, or even DAVector<DAVector<complex> for determining parameter dependence of dynamics parameters. In addition, our classes are templated on dimension and order (we did not discuss this in the paper), which allows for aggressive compile time optimisation and adds to efficiency. We hope that in the nearest future we can further develop the library by testing our ideas of optimisation for cache. We will have to add elementary functions and all needed operators in DA vector class, create the map library including normal forms, symplectification and Lie factorisation.

6 REFERENCES

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

[2] A. Dragt and M. Venturini, "Design of Optimal Truncated Power Series Algebra. Routines: II. Computing Sums and Ordinary and Lie Polynomials Using Monomial Indexing and Linked Lists," University of Maryland, Sept. 1996 (Draft).

[3] M. Nelson, C++ Programmer Guide to the Standard Template Library (IDG Books Worldwide, Foster City, CA, 1995).

[4] G.Furnish, "Container-Free Numerical algorithms in C++," Computers in Physics, v.12, No 3, 258(1998).

[5] John R. Cary and S. G. Shasharina, "Efficient C++ Library for Differential Algebra," Proceeding of European Particle Accelerator Conference, Stockholm 1998.

[6] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimisation of Blocked Algorithms," Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April, 1991.