EXPRESSION TEMPLATES FOR TRUNCATED POWER SERIES

John R. Cary and
Svetlana G. Shasharina, Tech-X Corporation, 4588 Pussy Willow Court, Boulder, CO 80301

Abstract

Truncated Power Series technique (Differential Algebra or DA) is a powerful tool for non-linear map analysis of accelerators. The most natural language for numerical DA's is C++, since it is object oriented and has operator overloading. Traditional C++, though, can be inefficient for scientific programming due to creation of many temporaries and extra loops in overloaded operators. Recent Expression Templates technique allows a user to combine the elegance of object oriented approach with the speed of procedural languages. The way it was created, it is not directly applicable for DA. We created a set of classes whose structure will be suitable for implementing DA vectors and maps. Classes realizing the Expression Templates technique are separated from the client classes, which allows their reuse for different mathematical concepts. Speed tests on KCC compiler showed that new C++ classes for DA have the same speed as hand-coded C.

1 PROBLEMS OF NUMERICAL IMPLEMENTATION OF DIFFERENTIAL ALGEBRA

Numerical differential algebra methods (see Ref. [1]) have found increasing use as they can be applied to arbitrary dynamical systems, for example accelerators or other beam and optics devices. The effect of passing of particles or rays through the system can be described mathematically by a map relating the final coordinates (coordinates in the most general sense: momentum, position, charge, mass etc.) to the initial. The map contains the information needed to evaluate such quantities of interest as nonlinear oscillation frequencies and chromatic aberrations. However, except for several trivial cases, it is impossible to find a closed mathematical solution for the map. But one can find solutions up to some order in distance from the central trajectory through perturbation theory based on power series expansion of the map. Numerical differential algebra methods permit one to carry out this perturbation theory for very high order maps, e.g., twelfth order in six variables, where there are millions of separate terms in the map.

For the purpose of this paper, one need to know only a couple of facts about DA vectors. In principle, it is an array of coefficients, obtained from Taylor expansion of functions in multidimensional space. These vectors can be added, subtracted, multiplied etc., and the result of most operations depends on the whole set of coefficients of operands. Thus, DA vectors can be represented by a vector-type class with overloaded operators. That is the reason why the implementation of DA methods is easiest within object oriented programming (see [2]), which provides a powerful triad: encapsulation, inheritance, and dynamic binding. C++ is especially well suited since it provides operator overloading.

Unfortunately, a big performance price must be paid to realize many of these benefits. Performance losses of a factor up to ten times (exact number depends on vector length) have been reported for codes rewritten in C++. The performance loss arises because usual C++ programming practices lead to the creation of many temporaries and the separation of complicated loops into loops with fewer operations. The latter is especially bad in current times, where RISC chips may have several arithmetic units that can be working simultaneously.

As a concrete example, we consider the evaluation of the addition among these vector objects in the following code fragment.

```
Vector y, a, b, c;
y = a + b + c;
```

With conventional Vector objects, usually defined as one-dimensional arrays, perhaps including array limits, with properly overloaded arithmetic operators, the typical C++ compiler generates code equivalent to:

```
int n = a.size();
Vector temp1 = new double[n];
for(int i=0; i<n; i++) temp1[i] =
a[i] + b[i];
Vector temp2 = new double[n];
for(int i=0; i<n; i++)
   Vector temp2[i] = temp1[i] + c[i];
for(int i=0; i<n; i++)
   y[i] = temp2[i];
delete [] temp1;
delete [] temp2;
```

The temporary storage variables temp1 and temp2 must each be created and destroyed. Indeed, the creation of temp2 is unecessary, but usually done by C++ compilers that are not optimized with regard to the return of user defined objects from operators and functions. In addition, there are three separate loops above, when only a single loop,

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

is needed. The loss of efficiency due to creation of temporaries and having separate loops can be very significant. Whenever users use C-style similar to the coding above, it is called hand-coded C.

0-7803-4376-X/98/$10.00 © 1998 IEEE

In the next section, we describe how new C++ technique, called Expression Templates (Ref. [3]) allowed us to make C++ classes for DA elements as fast as hand coded C.

2 EXPRESSION TEMPLATES FOR DA VECTORS

We start from giving the a partial implementation for the DA vector class. From the operational point of view, it has only addition operator (all the rest can be implemented in a similar way). For the sake of brevity, we declared everything public. We also had to acquire an awkward way of formatting the code in order to preserve the format of the paper:

```
class DA {
public:
  double* data;
  int length;
  DA(int n) : length(n){data = new
double[n];}
  DA(){}
  DA(const DA& x) {
      length = x.length;
      data = new double[length];
      for(int i=0;i<length;++i)
        data[i] = x.data[i];}
  ~DA() {delete [] data;}
//  Access
  double& operator[](int i)
      {return data[i];}
  const double& operator[] (int i)
     const {return data[i];}
//  Assignment to expression
  template <class A> DA& operator=
      (const DAExpr<A>& result){
      for (int i=0;i<length;++i)
        data[i] = result[i];
      return *this;}
//  Assignment to DA
  DA& operator = (const DA& a){
      if (this == &a) return *this;
      delete [] data;
      length = a.length;
      data = new double[length];
      for(int i=0;i<length;++i)
        data[i] = a.data[i];
      return *this;}
//  Summation
  template <class A, class B>
  static double Add
     (const A& a, const B& b, int i)
     {return a[i]+b[i];}
};
```

This class has typical (for vector-like classes) members: `double* data` and `int length`, describing the set of coefficients and their number. It has a constructor from `int`, a copy constructor and a typical destructor. In order to insure a normal behavior of assignment, we need to provide assignment to `DA` with deep copying (memory management with garbage collection can change its implementation). To implement inlining of all expressions containing vectors, we need assignment to DA expressions. Class for DA expressions is template and relies on behavior of `operator[](int)`, which is governed by the type of the template parameter `A`:

```
template<class A>
class DAExpr {
private:
  A iter;
public:
  DAExpr(const A& a) : iter(a){}
  double operator[](int i) const
      {return iter[i];}
};
```

Assignment to `DAExpr` is a template member function of a no-template class. Not so many compilers provide this capability: KCC compiler (Ref. [4]) is the only one we know. If the compiler does not have template members, see Ref. [3] for the trick

Consider how all this should work in a simple example:

```
DA y, a, b;
y = a + b;
```

Since this operation will call for assignment of y to an expression (a+b), summation of two DA vectors should return a DA expression, whose type is determined by the performed operation. In this case it is a binary operation of + type (`DApAdd`) between two double arrays:

```
DAExpr<DABinOp<double*,double*,DApAdd
<DA,double*,double*> > >
operator+(const DA& a, const DA& b)
{typedef DABinOp <
double*,double*,DApAdd<DA,double*,
double*> >  ExprT;
return DAExpr<ExprT>
(ExprT(a.data,b.data));}
```

When the resulting expression is assigned to a vector, `operator[](int)`, sitting in the assignment operator, applies this operator to `DAOpBin` (since it is the filling of `DAExpr` in this case). Inside `DAOpBin::operator[](int)`, function `Op::Apply` is called (with `Op` in our case being `DApAdd`)

```
template<class A, class B, class Op>
class DABinOp {
private:
  A iter1;
  B iter2;
public:
  DABinOp(const A& a, const B& b)
        : iter1(a), iter2(b) {}
```

```
    double operator [] (int i) const
    {return Op::Apply(iter1,iter2, i);}
};
```

Function `Op::Apply` in the case of addition is member of a separate class `DApAdd` (we have to have separate classes for all arithmetic operations):

```
template <class C,class A,class B>
class DApAdd {
public:
  DApAdd() { }
  static inline double Apply
     (const A& a, const B& b, int i)
     {return C::Add(a,b,i);}
};
```

It does not provide any specific behavior. It relies on the class type `DA` to know how to perform its operations by calling `C::Add(a,b,i)`, where `C` is `DA` in our example. We have to provide full arguments `iter1`, `iter2` and index i for `Op::Apply`, because in some cases (like multiplication and many other operations and functions for DA vectors) calculation of i-th component of the resulting vector involves several components of operands and depends on i.

In case of operation y = a + b + c, the compiler actually does (a+b)[i]+c[i] = a[i]+b[i]+c[i], so that the two loops are fused in one, which makes operations as fast as hand-coded C.

One actually must define four + operators: one for each of the possible combinations of `DAVec` and `DAVExpr`. None of these operators contain loops. The template mechanism combines all of the expressions together into one assignment, where the loop occurs. Thus, all of the loops always collapse to a single loop.

The novelties here compared with simple vector classes of Ref. [3], are:

(a) We pass whole objects and index to `Apply` function in order to generalize to more complicated structures. Correspondingly, we had to make class `DApAdd` template.

(b) We separated implementation of operations from the ET structure, putting it into the class itself. This idea is similar to *Glommables* of Ref.[4].

(c) We had to introduce + between 2 expressions in order to deal with situations like y=(a+b)+(c+d).

We performed speed test for our classes on KCC compiler. The results for time to perform one addition are shown on the figure. The upper curve corresponds to C++, another two are: C++ with Expression Templates and hand-coded C, which are practically indistinguishable. The speed enhancement over traditional C++ is enormous!

REFERENCES

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

[2] S. Lippman, *C++ Primer, second edition* (Addison-Wesley, Reading, Massachusetts, 1991).

[3] T. Veldhuizen, "Using C++ Template Metaprograms," C++ Report, vol.7, No. 4, 36 (1995).

[4] http://www.kai.com.

[5] G. Furnish, " Disambiguated Glommable Expression Template," preprint UCRL-JC-126523 (February, 1997, LLNL), submitted to Computers in Physics.