Tensor Contraction in C++

James F. Blinn

Microsoft Research

In my last column (January/February 2001 *CG&A*), I talked about a notational device for matrix algebra called tensor diagrams. This time I'm going to write some C++ code to symbolically evaluate these quantities. This gives me a chance to play with some as yet untried features in the C++ standard library, such as strings and standard template library (STL) container classes. (For more on STL, check out my favorite book on the subject: *The C++ Standard Library* by Nicolai M. Josuttis, Addison Wesley, Reading, Massachusetts, 1999.) I'm trying to figure out if I like these new-fangled programming tools by seeing how much code I can get away with not writing these days. My initial impression is positive. So this column is part demo, part advertisement of the STL's benefits. If you spend some time learning these tools, you too can write less code that does more.

The basic objects

Since this column is mostly about C++, I'll try to make it independent of my previous columns and introduce just enough algebra to motivate the code. We're interested in algebraic curves defined by $F(x, y, w) = 0$. The simplest of these is a straight line, represented by the equation

$$Ax + By + Cw = 0$$

We represent a quadratic curve with the equation

$$Ax^2 + 2Bxy + 2Cxw + Dy^2 + 2Eyw + Fw^2 = 0$$

and a cubic curve with the equation

$$\begin{aligned} Ax^3 &+ 3Bx^2y + 3Cxy^2 + Dy^3 \\ &+ 3Ex^2w + 6Fxyw + 3Gy^2w \\ &+ 3Hxw^2 + 3Jyw^2 \\ &+ Kw^3 = 0 \end{aligned}$$

(Note that I garbled this last equation in my previous column—see Figure 1a on page 86 in the January/February 2001 issue. This is the correct version.)

We want to relate various geometric properties of these curves with algebraic combinations of the coefficients A through K. First, we arrange the coefficients in arrays that, in mathematical lingo, are called tensors. For a line, we arrange the coefficients into a column vector:

$$\mathbf{L} = \begin{bmatrix} A \\ B \\ C \end{bmatrix}$$

For the quadratic, we define the symmetric matrix

$$\mathbf{Q} = \begin{bmatrix} A & B & C \\ B & C & E \\ C & E & F \end{bmatrix}$$

For the cubic curve equation, we arrange the coefficients into a $3 \times 3 \times 3$ tensor that we can (clumsily) write as a vector of matrices:

$$\mathbf{C} = \left[\begin{bmatrix} A & B & E \\ B & C & F \\ E & F & H \end{bmatrix} \begin{bmatrix} B & C & F \\ C & D & G \\ F & G & J \end{bmatrix} \begin{bmatrix} E & F & H \\ F & G & J \\ H & J & K \end{bmatrix} \right]$$

When we write an element of one of these tensors, the convention is to label its location in the array with indices written as subscripts. These are called covariant indices, and they look like L_i, Q_{ij}, C_{ijk}.

A point, on the other hand, is written as a row vector $\mathbf{P} = [x\, y\, w]$. When we write an element of the point, we label it with an index written as a superscript. (This doesn't mean exponentiation. It's just a place to put the index.) These indices are called contravariant and look like P^i.

Finally, we'll have use for a constant $3 \times 3 \times 3$ contravariant tensor called epsilon, whose elements we define as

$$\begin{aligned} \varepsilon^{123} = \varepsilon^{231} = \varepsilon^{312} &= +1 \\ \varepsilon^{321} = \varepsilon^{132} = \varepsilon^{213} &= -1 \\ \varepsilon^{ijk} &= 0 \quad \text{otherwise} \end{aligned}$$

The basic operation

The basic quantity we want to calculate is a "contraction" of two or more tensors. Vector dot products and

vector–matrix products are special cases of tensor contractions. More generally, we want to evaluate expressions such as

$$\sum_{ijklmn} \varepsilon^{ijk}\varepsilon^{lmn}Q_{il}Q_{jm}Q_{kn} \quad (1)$$

And being lazy, we'll abbreviate Equation 1 by leaving out the summation sign. We'll assume a summation over any index that appears exactly twice (once as covariant and once as contravariant) in any expression.

So, I want to come up with a simple symbolic algebra manipulation program that's good at these sorts of expressions and doesn't really need to handle anything more general. We saw last time that Equation 1 should evaluate to six times the determinant of **Q**. So if we translate Equation 1 into C++ code, we want the program to print out something like this:

$$6ADF + 12BCE - 6C^2D - 6AE^2 - 6B^2F \quad (2)$$

Incidentally, if this expression is zero it tells us that the quadratic is the product of two linear factors—that is, the equation stands for two straight lines.

The epsilon routine

The most obvious first step is to define a routine to calculate epsilon as follows:

```
int epsilon(int i, int j, int k)
{
  if(i==1 && j==2 && k==3) return 1;
  if(i==2 && j==3 && k==1) return 1;
  if(i==3 && j==1 && k==2) return 1;
  if(i==3 && j==2 && k==1) return -1;
  if(i==1 && j==3 && k==2) return -1;
  if(i==2 && j==1 && k==3) return -1;
  return 0;
}
```

We're ultimately going to define a class Expression to hold the final answer and a subroutine `Q(i, j)` that returns the symbolic value of element *i,j*. If we design these properly, we can write code to evaluate Equation 1 that simply loops through all values of *i*, *j*, *k*, *l*, *m*, and *n* and adds up the terms like so

```
#define forIndex(I) //
  for(int I=1;I<=3;++I)

Expression E;
forIndex(i)
forIndex(j)
forIndex(k)
forIndex(l)
forIndex(m)
forIndex(n)
  E += epsilon(i,j,k)*
       epsilon(l,m,n)*
       Q(i,l)*Q(j,m)*Q(k,n);
cout << E << endl;
```

This, however, gets out of hand pretty quickly. The epsilon tensor is mostly zeroes; in fact, only 6 out of the 27 entries are nonzero. If you have two epsilons in your expression, only 6*6 out of the 27*27 possible combinations are nonzero (about 1 in 20). Some of the diagrams we'll be ultimately interested in can have eight or more epsilons. This means that only 6^8 out of the 27^8 iterations (about 1 in 168,151) actually adds anything to `E`. And for each epsilon, you have three nested loops. There must be a better way.

There is. We'll turn the loops inside out. Instead of generating all combinations of indices, we'll have each loop go through the six nonzero epsilon values and return to us their indices and signs. The new epsilon function looks like

```
int epsilon (int which,
  int* pI, int* pJ, int* pK)
{
static int Ix1[6]={1,2,3, 3, 1, 2};
static int Ix2[6]={2,3,1, 2, 3, 1};
static int Ix3[6]={3,1,2, 1, 2, 3};
static int Sgn[6]={1,1,1,-1,-1,-1};
  *pI=Ix1[which];
  *pJ=Ix2[which];
  *pK=Ix3[which];
  return Sgn[which];
}
```

The loop to evaluate Equation 1 looks like

```
#define forEpsilon(e) //
  for (int e=0; e<6; ++e)

Expression E;
forEpsilon(e1)
forEpsilon(e2)
{
  int i,j,k,l,m,n;
  int sign=epsilon(e1,&i,&j,&k)
           *epsilon(e2,&l,&m,&n);
  E +=sign*Q(i,l)*Q(j,m)*Q(k,n);
}
cout << E<<endl;
```

Note that I broke the summation statement into two, because the calls to `epsilon` return the index values and must be executed before the calls to `Q`. Putting these into the same statement might work but it's a bit dicey. The code as shown guarantees correct operation.

The Expression object

Now let's look at what we must do, in C++ terms, to make the statement

```
E +=sign*Q(i,l)*Q(j,m)*Q(k,n);
```

make sense. The result we expect from this, Equation 2, is a sum of five terms, each of which is some integer times the product of three elements of the **Q** matrix. So, program-wise, an `Expression` is some sort of list of `Terms`. And each `Term` consists of an integer factor that

I'll call **Ifactor** and some sort of list of variable-name factors that I'll call **Vfactors**. There are many ways to manage lists in C++, but to see which one is best, we must look at how we'll use the lists.

What must happen when the statement executes? The code for the **operator+=** must search through the existing elements of **E** to see if there is already one there that has the same **Vfactors** as the **Term** being added. If there is, it adds **sign** to the **Ifactor** field of the **Term**. If such a **Term** doesn't exist, we must insert one and initialize the **Ifactor** field to **sign**. So, what are the basic operations we'll be doing a lot of? We'll search the **Expression** list for an entry containing the desired **Vfactors**, which implies doing a lot of comparisons between various **Vfactors** lists.

My first design decision, then, is to only allow single characters for symbolic variables and to make the **Vfactors** list be a C++ standard **string**. A simple, built in, string comparison can then compare two **Vfactors** lists.

Now, what kind of list should we use for the **Expression** object? My first try (as all first tries should be) was to use an STL **vector** and all the standard STL operations for searching and inserting into it. Then, after reading a bit further in the STL manual, I found another collection object that's better suited for the **Expression** class—it's called a **map**. A **map** is a collection of key–value pairs that are kept sorted on the key for easy lookup. We'll make the key the **Vfactors** string and the value field the **Ifactor** integer. That is, instead of making **Expression** be an explicit list of **Terms** we'll make it a map from **Vfactors** values to **Ifactor** values. What's really nice about the STL implementation of **map** is that we can syntactically access it as though it were an associative array; the subscription operator is overloaded to do a lookup (and insertion if necessary) and return a reference to the appropriate value field. The whole **Expression** class becomes almost trivial:

```
struct Term
{
  int    Ifactor;
  string Vfactors;
// definition shown below
};

struct Expression
{
  map<const string,int>TermList;

  Expression&
  operator+=(const Term& T)
  {
  TermList[T.Vfactors]+=
                 T.Ifactor;
  return *this;
  }
};
```

That's really all there is to it. The C++ compiler provides anything else you need by default. (I know I'm bad for making these **struct** instead of **class**. In the real world, you should provide accessor functions for all internal variables and make the variables private. Doing so here would clutter up the code and obscure some of my main points.) For printing purposes, we add an inserter operator that uses the standard STL mechanism for iterating through the map:

```
ostream&
operator<< (ostream& out,
    const Expression& E)
  {
  map<const string, int>::
    const_iterator i;
  for(i =E.TermList.begin();
      i!=E.TermList.end(); ++i)
      out <<showpos<< i->second
     <<" "<< i->first;
  return out;
  }
```

The Term object

Now we need to gen up some arithmetic operators that will allow the C++ expression

```
sign*Q(i,l)*Q(j,m)*Q(k,n)
```

to construct the appropriate **Term** object to pass to **Expressions' operator+=**. Recall that the variable **sign** is the integer result of multiplying several calls to **epsilon**. Simply having **Q** return a single **char** won't work because C++ is perfectly happy to add **ints** and **chars** as numeric quantities. No, we must have a user-defined class **Symbol** that wraps the return from **Q** and lets us define some multiplication operators within **Term** that accept **ints** and **Symbols**:

```
class Symbol
{
  char c;
public:
  Symbol(const char ci):c(ci) {}
  char name() const {return c;}
};
```

The automatic conversion from **char** to **Symbol** lets us write the **Q** routine simply

```
Symbol Q(int i, int j)
{
  static char V[]="ABCBDECEF";
  return V[ (i-1)*3+(j-1) ];
}
```

Next, we make a **Term** constructor that will convert the integer **sign** into a **Term** with a null **Vfactors** string. Then, we make a multiplication operator for **Term*Symbol** that simply takes the character from **Symbol** and appends it to the **Vfactors** string. Finally, to make these strings mathematically comparable by doing a string comparison, we'll keep the **Vfactors** sorted. Fortunately, there's a handy algorithm in STL

that makes this easy. The final **Term** class looks like

```
struct Term
{
  int    Ifactor;
  string Vfactor;

  Term(int s): Ifactor (s),
    Vfactor ( ) {;}

  Term&
  operator*=(const Symbol& s)
  {
    Vfactor += s.name();
    sort(Vfactor.begin(),
      Vfactor.end  ());
    return *this;
  }
};

const Term
operator*(const Term& lhs,
    const Symbol& S)
  {return Term(lhs) *= S;}
```

It works

That's all there is to it. We just mash this together with the header files,

```
#include <string>
#include <iostream>
#include <algorithm>
#include <map>

using namespace std;
```

and the code prints out the desired result:

```
+6 ADF-6 AEE-6 BBF+12 BCE-6 CCD
```

Close enough. I can handle rewriting this using exponents myself.

Examples

Now let's play with this. We first define some more constant tensors:

```
////  LINES /////
Symbol L(int i)
{
  static char V[]="abc";
  return V[i-1];
}
Symbol M(int i)
{
  static char V[]="def";
  return V[i-1];
}
///  Arbitrary Transformation
Symbol T(int i, int j)
{
  static char V[]="abcdefghj";
  return V[ (i-1)*3+(j-1) ];
}
/// GENERAL CUBIC CURVE ///
char C(int i, int j, int k)
{
  static char V[]=
  "ABEBCFEFHBCFCDGFGJEFHFGJHJK";
  return V[(i-1)*9+(j-1)*3+(k-1)];
}
```

Cross product

The original motivation for defining epsilon was as an abbreviation for the cross product. Here, we verify that it works by evaluating $C^k = \varepsilon^{ijk} L_i M_j$. The cross product has one free index (k here) so it necessitates use of an array of **Expression** objects where:

```
Expression C[3];
forEpsilon(e)
{
  int i,j,k;
  int sign = epsilon(e,&i,&j,&k);
  C[k-1] += sign*L(i)*M(j);
}
forIndex(k)
  cout <<"C("<<k<<")="<<
    C[k-1]<<endl;
```

Line quadratic tangency

The condition that a line L is tangent to a quadratic curve Q is $\varepsilon^{ijk}\varepsilon^{lmn} Q_{il} Q_{km} L_j L_n = 0$:

```
Expression E;
forEpsilon(e1)
forEpsilon(e2)
{
int i,j,k,l,m,n;
  int s = epsilon(e1,&i,&j,&k)
        * epsilon(e2,&l,&m,&n);
  E += s*Q(i,l)*Q(k,m)*L(j)*L(n);
}
cout << E<<endl;
```

An epsilon identity

Here's a confirmation of an identity that will become important later: applying three copies of a transformation matrix to the three indices of **epsilon** results in a bare epsilon times the scalar det**T**.

$$\varepsilon^{ijk} T_i^l T_j^m T_k^n = \varepsilon^{lmn} \left(\det \mathbf{T} \right)$$

This expression has three free indices (l, m, and n), but we can do this one without an array of Expressions:

```
forIndex(l)
forIndex(m)
forIndex(n)
{
  Expression E;
  forEpsilon(e1)
    {
      int i,j,k;
```

```
      int s=epsilon(e1,&i,&j,&k);
      E += s*T(i,l)*T(j,m)*T(k,n);
    }
  cout <<l<<m<<n<<" "<<E<<endl;
}
```

Some cubic identities

The following quantity is identically zero for all values of **C**:

$$Z_l^k = \varepsilon^{ijk} C_{ijl}$$

We verify this by

```
forIndex(l)
{
  Expression E[3];
  forEpsilon(e1)
  {
    int i,j,k;
    int s = epsilon(e1,&i,&j,&k);
    E[k-1] += s * C(i,j,l);
  }
  forIndex(k)
    cout <<"Z("<<l<<k<<")="
      <<E[k-1]<<endl;
}
```

This means that if we see such an expression embedded in a larger expression, we can immediately say that the whole thing is zero.

Likewise, the following expression containing a cubic is also identically zero for any tensor **C**.

$$\varepsilon^{ijk}\varepsilon^{lmn}\varepsilon^{pqr} C_{ilp} C_{jmq} C_{knr}$$

```
Expression E;
forEpsilon(e1)
forEpsilon(e2)
forEpsilon(e3)
{
  int i,j,k, l,m,n, p,q,r;
  int s = epsilon(e1,&i,&j,&k)
        * epsilon(e2,&l,&m,&n)
        * epsilon(e3,&p,&q,&r);
  E +=s*C(i,l,p)*C(j,m,q)*C(k,n,r);
}
cout <<E<<endl;
```

The final example is a bit more complex. For reasons that I'll explain in a later column, I call it the "cube invariant":

$$\varepsilon^{ijk}\varepsilon^{lmn}\varepsilon^{pqr}\varepsilon^{tuv} C_{nqv} C_{jru} C_{kmt} C_{ilp}$$

I'm going to do something slightly different this time for the C++ code. All our examples so far have made the code mimic the algebraic expression. This is easy to use but can be a bit slow. That's because there's a lot of creation, copying, sorting, and destruction of temporary `Term` variables during their execution. I'll write the code for this example to show an alternate way to generate the `Term` that uses our existing machinery but avoids unnecessary creation and deletion:

```
Expression E;
forEpsilon(e1)
forEpsilon(e2)
forEpsilon(e3)
forEpsilon(e4)
{
  int i,j,k, l,m,n, p,q,r, t,u,v;
  Term T(epsilon(e1,&i,&j,&k)
       * epsilon(e2,&l,&m,&n)
       * epsilon(e3,&p,&q,&r)
       * epsilon(e4,&t,&u,&v));
  T *= C(  n,q,v);
  T *= C(j  ,r,u);
  T *= C(k,m,  t);
  T *= C(i,l,p  );
  E += T;
}
cout << E<<endl;
```

How do I like this?

C++ giveth and C++ taketh away. The internal machinery of the map object keeps its key–value pairs stored in some sort of binary tree thingy so that searching is very fast. This is something I wouldn't have felt like getting into myself. That's good. But the simple mimicking of an algebraic expression by a C++ expression implies a lot of strange jiggery pokery going on during the creation of each `Term`. That's not so good. But who cares really. The execution time of the examples shown here is negligible. It's only when you get upwards of eight epsilons that the program takes a noticeable amount of time. Rewriting the expression as in the final example helps here. For even more complex situations, it would be easy to add an explicit `Term` constructor that's passed a sign and some fixed number of `Symbols`. This constructor could resize `Vfactors` once, explicitly assign the `Symbols` to it, and only need to sort it once.

What's next

One can imagine any number of extensions to this code to handle more general expressions. But what we have here is good enough to serve as a check on our theoretical investigations to make sure we don't miss any constant factors or stray minus signs. I'll post the source code for these examples on my Web site at http://www.research.Microsoft.com/~blinn.

Now that we can verify some of our computations, in my next column I'll define and calculate some algebraic quantities that are invariant under coordinate transformation. In particular, there's an interesting geometric meaning to the quantity calculated by our final example. ■

Readers may contact Blinn by email at blinn@microsoft.com.