uBlas: Boost High Performance Vector and Matrix Classes

Juan José Gómez Cadenas

University of Geneve and University of Valencia

(thanks to:

Joerg Walter, uBlas co-author.

Todd Vedhuizem, ET co-inventor)

Vector and Matrix classes in C++

- Use of C++ vector and matrix classes for scientific calculations typically results in poor performance w.r.t Fortran or C. This is due two several factors:
  - Use of virtual functions (dynamic polymorphism)
  - Temporaries

Polymorphism

- Standard tool in C++
- Requires virtual functions that have big performance penalties
  - Extra memory access
  - Compiler cannot optimize around the virtual function call. It prevents desired features such as loop unrolling, etc.
- Virtual functions are acceptable if function is big or not called very often

Polymorphism (II)

- Unfortunately, in scientific code some of the most useful places for virtual functions are in inner loop bodies and involve small routines

```
class HepGenMatrix {
public:
virtual ~HepGenMatrix() {}
virtual int num_row() const = 0;
virtual int num_col() const = 0;
virtual const double & operator()(int row, int col) const =0;
virtual double & operator()(int row, int col) =0
```

Virtual function dispatch to operator () results in poor performance

Static Polymorphism

- Replace dynamic polymorphism with static (i.e, compile time) polymorphism
- Use of expression templates
- Expression templates heavily depend on the famous Barton-Nackman trick, also coined 'curiously defined recursive templates'

Barton-Nachman trick

```
template class<T_leaf>
  class Matrix{
    public:
     T_leaf& assign_leaf(){
      return static_cast<T_leaf>(*this);}

     double operator () (int i, int j){   //delegate to leaf
      return assign_leaf()(i,j)
…
class symmetric_matric : public Matrix<symmetric_matrix>
```

Static Polymorphism at Work

- The trick is that the base class takes a template parameter which is the type of the leaf class. This ensures that the complete type of an object is known at compile time. No need for virtual function dispatch
- Methods can be selectively specialized in the leaf classes (default in the base, overridden when necessary)
- Leaf classes can have methods which are unique to the leaf class

Temporaries

When you write:

```
Vector a(n), b(n), c(n);
   a = b + c + d;
```

The compiler does the following:

```
Vector* _t1 = new Vector(n);
      for(int i=0; i < n; i++)
                _t1(i) = b(i) + c(i);

Vector* _t2 = new Vector(n);
      for(int i=0; i < n; i++)
                _t2(i) = _t1(i) + b(i);
```

Temporaries(II)

```
for(int i=0; i < n; i++)
        a(i) = _t2(i) + _t1(i) ;
delete _t2;
delete _t1;
```

So you have created and deleted two temporaries!

Performance Implications

- For small arrays (HEP case!) the overhead of new and delete result in very poor performance (about 1/10 of C)
- For large arrays the cost is in the temporaries. It depends on the operation. For example, they are expensive for + operation

Expression Templates

- Invented independently by Todd Veldhuizen and Daveed Vandevoorde
- The basic idea is to use operator overloading to build parse trees.
- Take advantage of the basic fact that a class can take itself as a template parameter

Example

Array A,B,C,D;

D=A+B+C;

The expression A+B+C could be represented by a type such as:

X<Array, plus, X<Array, plus, Array>>

Consider:

struct plus{} ; // addition

class Array {}; // some array class

Example (cont)

```
// X represents a node in a parse tree
template<typename Left, typename Operation, typename Right>
class x{};

//The overloaded operator with does parsing for expressions of the
// form A+B+C+D…
Template<class T>
X<T, plus, Array> operator + (T, Array){
    return x<T, plus, Array> ();
}
```

Example (cont)

With the above code, A+B+C is parsed like this:

```
Array A,B,C,D;
D=A+B+C;
X<Array, plus, Array> ()+ C;
=X<X<Array, plus, Array>, plus, Array> ();
```

ET: Minimal implementation

Of course to be useful you need to store data in the parse tree (e.g. pointers to the arrays).

Here is a minimal expression templates implementation for 1-D arrays. First, the `plus` function object:

```
struct plus {
public:
    static double apply(double a, double b) {
        return a+b;
    };
};
```

The parse tree node:

```
template<class T_op, class T1, class T2>
struct X {
    T1 leftNode_;
    T2 rightNode_;

    X(T1 t1, T2 t2)
        : leftNode_(t1), rightNode_(t2)
    { }

    double operator[](int i)
    { return T_op::apply(leftNode_[i],rightNode_[i]); }
};
```

Now the array class:

```
struct Array {
    Array(double* data, int N)
        : data_(data), N_(N)
    { }

    // Assign an expression to the array
    template<class T_op, class T1, class T2>
    void operator=(X<T_op,T1,T2> expression)
    {
        for (int i=0; i < N_; ++i)
            data_[i] = expression[i];
    }

    double operator[](int i)
    { return data_[i]; }

    double* data_;
    int N_;
};
```

And the operator+:

```
template<class T>
X<plus, T, Array> operator+(T a, Array b)
{
    return X<plus, T, Array>(a,b);
}
```

See the loop being built step by step:

```
D = A + B + C;
  = X<plus,Array,Array>(A,B) + C;
  = X<plus,X<plus,Array,Array>,Array>(X<plus,
        Array,Array>(A,B),C);
```

Then it matches to template `Array::operator=`:

```
D.operator=(X<plus,X<plus,Array,Array>,
    Array>(X<plus,Array,Array>(A,B),C) expression)
{
   for (int i=0; i < N_; ++i)
     data_[i] = expression[i];
}
```

See how expression[i] is evaluated by `X::operator[]`:

```
     data_[i] = plus::apply(X<plus,Array,
                   Array>(A,B)[i], C[i]);
              = plus::apply(A[i],B[i]) + C[i];
              = A[i] + B[i] + C[i];
```

uBlas

- Consistent use of expression templates to eliminate virtual function calls and temporaries results in very high performance (for a C++ standalone library)
- Carefully designed (boost pair reviewed) interface. Maps Blas calls
- supports conventional dense, packed and basic sparse vector and matrix storage layouts
- Symmetric, hermitian, triangular matrices, etc
- Template type (T=int, float, double, complex…)
- STL like iterators
- Proxies (ranges, slices) to access views of vector and matrices

uBlas (ii)

- Extensive checking via consistent use of exceptions
- Very well documented
- Part of the boost library (i.e, reliable maintenance)

uBlas (III)

- Real High Performance libraries (like ATLAS) are using platform specific assembler kernels
- Toon Knapen and Kresimir Fresl are working on C++ bindings to such kernels, which already allow the interfacing of uBLAS with ATLAS

Comments on CLHEP matrix classes

- 10 years old already (i.e, a success!)
- But:
  - Use of virtual functions
  - Inefficient array indexing M[][] (temp objects)
  - Temporaries problems
  - “Messy” interface
    - Linear algebra functions are often part of the class
    - M.inverse()???

Conclusion

- uBlas: Modern C++, very professional, very well documented, part of boost.
- Fast
- “Blas compliant”
- Very clean interface

- Seems a very good candidate to replace current CLHEP vector and matrix classes