Implementation of Linear Algebra Packages in C++

Michael Lehn

Preprint Series: 2005-05

Fakultät für Mathematik und Wirtschaftswissenschaften
UNIVERSITÄT ULM

Implementation of Linear Algebra Packages in C++

Micahel Lehn

Preprint Series: 2005-05

Fakultät für Mathematik und Wirtschaftswissenschaften
UNIVERSITÄT ULM

Submitted to: C/C++ Users Journal on 20 Mar 2005

Author: Michael Lehn

==========

Implementation of Linear Algebra Packages in C++

Abstract

C++ provides capabilities for code reuse through abstraction, generic programming and expressiveness through operator overloading. Even though these features make C++ interesting for scientific computing it has not been generally adopted in this field. This is mainly due to the lack of performance that arise when some of these features are not used with special caution.

In this article I want to outline a C++ implementation of a linear algebra package that aims to overcome these shortages. The package intends to be applicable in teaching as well as research. While the first target requires providing expressive notations for linear algebra expressions the second one demands their efficient evaluation. Further, such a realization has to provide feasible ways to integrate new matrix/vector types. For this reason we are interested in a flexible mechanism for the evaluation of linear algebra expressions.

As a proof of concept I outline how such a package can be implemented using BLAS (Basic Linear Algebra Subprograms, [1]) for the evaluation. The BLAS provide 'building blocks' to perform matrix/vector operations. Hardware vendors provide highly optimized implementations for their particular platforms. Hence using these routines for the supported matrix/vector types provides the highest possible performance on a given platform in most cases.

0) Introduction

For a linear algebra package it is crucial that operations are evaluated as
efficient as possible. From this, several demands arise for an implementation:
+ Minimize the cost of temporaries;
+ Minimize copying of matrices and vectors;
+ Minimize multiple loops over the same data in composite operations;
+ Exploit the provided hardware.

For a few often-used linear algebra expressions one can provide particular functions that achieve these aims. As an example, consider the expression z = A*x+y, where x, y and z are vectors and A is a matrix. Calling a function mulAddAndAssign(&z, &A, &x, &y) will not

create any temporaries if coded properly. The remaining points listed above can be satisfied through an adequate implementation of this function. But obviously this approach is only feasible to support a limited number of expressions. This concept is realized by BLAS. For particular matrix-vector products the BLAS function gemv and for the sum of vectors the BLAS function axpy can be used. Compound expressions like in the above example can be computed by sequential use of such elementary functions. However this leads to a loss of expressiveness or even error prone code when expressions become more complex.

In C++ operator overloading can be used to achieve more expressiveness. But then special care has to be taken regarding the creation of temporaries. This is due to the fact that only unary and binary operators can be overloaded. Again consider the above expression and assume that an operator for the matrix-vector product and another for the sums of vectors are defined. Then temporaries for A*x and A*x+y are created and the latter is copied to z. In [2] this issue was addressed and a technique was outlined to overcome this problem. The technique uses the concept of closures from functional programming. However, the realization was only intended to support a limited number of expressions.

While my approach follows this concept I in particular focus on a realization that ideally can handle arbitrary complex expressions. Analogously to the BLAS concept only a small set of functions for some basic linear algebra operations has to be implemented. My linear algebra library implements a strategy independent of concrete matrix/vector types to efficiently evaluate complex expression using these functions.

I will first outline the design and implementation of matrix/vector types. Then I illustrate how for linear algebra expressions closures a concept from functional programming can be implemented. Therefore I use a technique which became known as expression templates ([3], [4], [5]). I conclude with the introduction of a simple mechanism that allows using BLAS functions as backend for the evaluation of closures.

1) Design and Implementation of Matrix/Vector Types

For a linear algebra package abstract matrix and vector types are the key to code reuse. Any algorithm written for abstract matrix/vector types can then be used for arbitrary implementations of these types. C++ realizes abstract types through polymorphism. In a base class merely the interface of a type is defined. Implementations of a concrete type are provided through derived classes. Usually polymorphism implies virtual functions, which can lead to serious performance problems. This can be avoided by a technique that became known as the Barton-Nackman-Trick [6], which is illustrated in the following for the matrix hierarchy:

```
template <typename Impl>
class Matrix
{
    public:
        Impl &
        impl()
        {
            return static_cast<Impl>(*this);
        }

        double &
        operator(int i, int j)
        {
            impl()(i, j);
        }

        // ...
};

class DenseMatrix : public Matrix<DenseMatrix>
{
    public:
        double &
        operator(int i, int j);   // Element access

        // ...
};

class SymmetricMatrix : public Matrix<SymmetricMatrix>
{
    // ...
};
```

The type of the derived class is provided to the base class through a template parameter. Calling a method in the base class results in conversion to the derived class and a subsequent call of the specialized version. Note that this can be inlined by a compiler as the type of the derived class is know at compile time and thus specialized methods are directly called.

The derived matrix classes can be further parameterized. For instance, the implementation of a banded matrix could contain template parameters for

+ the element type,
+ row or column oriented storage,
+ full, banded or packed storage.

An ellegant pattern for an implementation is given by the "simple engine" approach from the POOMA team ([4]). The concrete data representation hereby acts as an engine.

2) Closures for Linear Algebra Operations

Again consider the expression z = A*x+y. The idea of a closure is that the involved operators do not immediately perform the operations and subsequently return the results. Instead, the right hand side expression A*x+y gets transformed into a single object that contains references to the three operands and whose type reflects the whole expression. Such an object is called a closure and can be considered graphically as the representation of an expression tree:

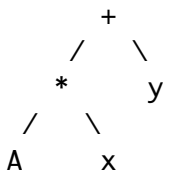

Fig.1: Expression tree for A*x+y

Thus the evaluation is deferred to the assignment operator that receives the closure.

The mechanism I use to transform expressions into closures is rather simple. Starting point therefore is the definition of closure classes representing binary operations. Depending on the resulting type after evaluation, these can be regarded as either specialized matrix or vector types. Hence I define two closure classes MatrixClosure and VectorClosure which are derived from Matrix and Vector respectively. They are further template parameterized with respect to operation and operand types. The closure objects only keep references to their two operands:

```
template <typename OP, typename L, typename R>
struct VectorClosure : Vector<VectorClosure<OP, L, R> >
{
    const L &l;
    const R &r;

    VectorClosure(const L &ll, const R &rr) : l(ll), r(rr);
};
```

Analogously the type MatrixClosure gets defined. Operand types are defined through a set of empty auxiliary classes, which simply act as type tags:

```
struct Mult {};
struct Add {};
// ...
```

Operators for linear algebra operations merely instantiate and return closure objects and can be implemented for abstract matrix/vector types:

```
template <typename M, typename V>
const VectorClosure<Mult, Matrix<M>, Vector<V> >
operator*(const Matrix<M> &A, const Vector<V> &x)
{
    return VectorClosure<Mult, Matrix<M>, Vector<V> >(A, x);
}
```

With other operators defined analogously an expression like A*x+y gets automatically transformed into one single closure object. Assume A is of type DenseMatrix and x, y are both DenseVector types. Then first the operator for the matrix-vector product gets called with A and x as arguments and returns a closure of type

```
VectorClosure<Mult,
              Matrix<DenseMatrix>,
              Vector<DenseVector>
             >
```

Second the operator for vector sums is called with this closure object and y as arguments and returns a closure object whose type represents the expression tree illustrated in Fig.1:

```
VectorClosure<Add,
              Vector<VectorClosure<Mult,
                                   Matrix<DenseMatrix>,
                                   Vector<DenseVector>
                                  >,
              Vector<DenseVector>
             >
```

The closure object further contains references to all three operands A, x and y. Hence all information needed to evaluate the expression is accessible through one object. Such a closure where at least one operand is again a closure a is in the following called a composite closure.

Unary operations like for instance the transposition of a matrix can be represented by instances of the same closure types. In this case both operands of the closure reference the same object. Appropriate operand tags obtain distinction of unary and binary closures:

```
struct Transpose {};

template <typename M>
const MatrixClosure<Transpose, Matrix<M>, Matrix<M> >
transpose(const Matrix<M> &A)
{
```

```
        return MatrixClosure<Transpose, Matrix<M>, Matrix<M> >(A, A);
    }
```

Regarding the implementation of the closure classes it is worth mentioning that operands must only be referenced if they live until the whole expression has been evaluated. This only can be guaranteed if the operands are not built-in C++ types as are for instance the user defined matrix/vector types. For operands that are scalar types the closure instead has to store a copy. This behavior can be achieved through an auxiliary trait class (see [5]).

3) Evaluation of Closures

Once the right hand side of an expression has been transformed into a closure object it is passed to an assignment operator of either a matrix or vector class. There a process is triggered that evaluates the closure and assigns the result to the left hand side.

A recursive scheme is followed to evaluate composite closures: first the operands are evaluated and afterwards the operation encapsulated by the closure is performed and the result assigned to the left hand side. Thus the evaluation of a complex expression is reduced to the evaluation of simpler expressions. In order to terminate the recursion one needs to provide functions for the evaluation and assignment of expressions that are "simple enough". Simple enough hereby roughly means that an expression can be evaluated through a single BLAS function.

3.1) Evaluation and Assignment of Composite Closures

I outsource the evaluation of closures into a set of overloaded assign functions. For vector closures the functionality of these functions corresponds to the general form

```
y <- alpha * [..] + beta*y
```

With [..] I denote an arbitrary vector closure and alpha, beta are scalars. Assignment, plus-assignment and minus-assignment operators in the abstract vector class fall back on calling an assign function with adequate parameters. In derived classes the copy constructor calls the assignment operator of the base class. The adequate parameters for instance are alpha=1, beta=0 for the assignment operator and alpha=1, beta=1 for the plus-assignment operator.

The assign functions for the closures are overloaded for the different operation types. Let me illustrate this for vector closures representing the sum of two vectors. I intend to reduce an assignment z=x+y to an assignment z=x and a subsequent update z+=y. If x or y are closures this leads to the evaluation of two simpler closures. Otherwise I require that these assignments can be performed for the concrete vector types. This concept is realized by calling assign

functions for the operands. Addition is obtained by providing adequate parameters:

```
template <typename E, typename L, typename R>
void
assign(Vector<E> &lhs,
       const Vector<VectorClosure<Add, L, R> > &rhs,
       double alpha, double beta)
{
    assign(lhs, rhs.impl().left(), alpha, beta);
    if (beta==0) { beta=1; }
    assign(lhs, rhs.impl().right(), alpha, beta);
}
```

While the scaling parameter alpha is only passed through the update parameter beta needs some special attention. If the initial value of beta is 0 I have an assignment like in z=x+y otherwise an update like in z+=x+y. In the first case I only want an update for the second operand and thus beta is changed to 1. For matrix closures assign functions are of similar form but also incorporate the complex or real transposition of matrices:

```
B <- alpha*op([..]) + beta*B
```

Here [..] indicates a matrix closure, op(A) denotes A, A^T or A^H.

For closures representing multiplications the implementations of assign functions are slightly more complicated. Considering expressions like (A+AT)*x or A*(x+y) I demand that the assign function first evaluates its operands and then performs the multiplication. This implies creation temporaries for operands that are closures. For this purpose I define the auxiliary trait class Result. I first illustrate its functioning and suspend the implementation details for a moment: if T is the type of a closure object then Result<T>::Type denotes the result type of the closure otherwise it denotes T itself. With this tool I can enforce the evaluation of closure objects:

```
enum Op {none=0, transposed=1, conjugate=2, conjugateTransposed=4};

template <typename E, typename L, typename R>
void
assign(Vector<E> &lhs,
       const Vector<VectorClosure<Mult, L, R> > &rhs
       double alpha, double beta,
       Op opA)
{
    const typename Result<L>::Type &A = rhs.impl().left();
    const typename Result<R>::Type &x = rhs.impl().right();
    assignProduct(A, x, alpha, beta, opA);
}
```

If an operand is not a closure object it is only referenced. Otherwise I enforce its evaluation and receive a temporary storing the result. The operation and assignment of the result is then performed by an external function assignProduct.

3.2) Evaluation and Assignment of Simple Expressions

The actual computation takes place in assignProduct functions and through calls of assign functions with non-closures as argument. For matrix/vector types supported by BLAS implementation of these functions represent merely a wrapper to corresponding BLAS functions. For other types I assume that it is possible to provide implementations of the same functionality.

I roughly outline a BLAS based implementation of some of these functions. For dense vectors the assignment can be implemented using the BLAS functions copy, scale and axpy:

```
void
assign(Vector<DenseVector> &lhs,
       const Vector<DenseVector> &rhs,
       double alpha, double beta)
{
    if (beta==0) {
        // copy
        if (alpha!=1) {
            // scale
        }
    } else {
        // axpy
    }
}
```

For a dense matrices and a dense vectors the matrix-vector product can be implemented by using the BLAS function gemv:

```
enum Op {none, transposed, conjugateTransposed};

void
assignProduct(Vector<DenseVector> &lhs,
              const Matrix<DenseMatrix> &A,
              const Vector<DenseVector> &x,
              double alpha, double beta,
              Op opA)
{
    // gemv
}
```

For the example z = A*x+y I can now illustrate the mechanism. With

boxed expressions I indicate the corresponding closure objects:

```
     Call of assign(z, A*x+y, 1, 0)
     ->   Call of assign(z, A*x, 1, 0)
          ->   Call of assignProduct(z, A, x, 1, 0)
     ->   Call of assign(z, y, 1, 0).
```

If as underlying implementation BLAS functions are used this is equivalent to

```
  +   Compute z = A*x using gemv
  +   Compute z = z + y using axpy
```

The Auxiliary Trait Class Result

We saw that for the treatment of products it is required to determine the resulting type of a closure type. For this purpose a trait class Result can be implemented. Starting point is hereby the definition

```
     template <typename A>
     struct Result
     {
         typename A Type;
     };
```

which gets specialized. Specialization hereby can be viewed as a rule stating the result type of a closure. For example specializations with only non-closure types like

```
     struct Result<VectorClosure<Add, DenseVector, DenseVector> >
     {
         typename DenseVector Type;
     };

     struct Result<VectorClosure<Mult, DenseMatrix, DenseVector> >
     {
         typename DenseVector Type;
     };
```

state that the sum of two dense vectors is again a dense vector and the product of a dense matrix with a dense vector a dense vector. Such rules are also crucial for the integration of new matrix/vector types. It for instance allows to define that multiplication of a sparse matrix with a dense vector results in a dense vector.

Through a recursive scheme these rules are used to determine the resulting type of a composite closure. For vector closures such a scheme is given by:

```
     template <typename I>
     struct Result<Vector<I> >
```

```
{
    typedef typename Result<I>::Type Type;
};

template <typename Op, typename I1, typename I2>
struct Result<VectorClosure<Op, I1, I2> >
{
    typedef typename Result<I1>::Type L;
    typedef typename Result<I2>::Type R;

    typedef typename Result<VectorClosure<Op, L, R> >::Type Type;
};
```

The first specialization defines the derived type of the vector as its resulting type. The second specialization first determines the type of its two operands and finally the resulting type of the closure. For the later it falls back on a specialization with only non-closure types like we saw above. For matrix closures a corresponding scheme can be achieve analogously by replacing "Vector" with "Matrix".

References

[1] BLAS, http://www.netlib.org/blas/.
[2] Bjarne Stroustrup, The C++ Programming Language, Addison Wesley, 1997.
[3] Todd L. Veldhuizen, Expression templates, C++ Report, 1995.
[4] Todd L. Veldhuizen,
Techniques for Scientific C++,
http://osl.iu.edu/~tveldhui/papers/echniques/, 1998.
[5] David Vandevoorde, Nicolai M. Josuttis,
C++ Templates - The Complete Guide, Addison Wesley, 2003.
[6] J. Barton, L. Nackman, Scientific and engineering C++, Addison Wesley, 1994.