Comput Visual Sci (2010) 13:59–68
DOI 10.1007/s00791-009-0128-2

REGULAR ARTICLE

Advanced expression templates programming

J. Härdtlein · C. Pflaum · A. Linke · C. H. Wolters

Received: 4 December 2006 / Accepted: 21 May 2008 / Published online: 16 November 2009
© Springer-Verlag 2009

Abstract Expression Templates (ET) are a powerful tool for development of user-friendly numerical libraries. By this concept and by operator overloading in C++, numerical algorithms can be implemented in a mathematical notation without decreasing the performance in comparison to optimized C or FORTRAN codes. In this paper, we present new Expression Template techniques. First, we explain the concept of "Easy Expression Templates", which are easier to implement than classical ET. Then, we explain "Fast Expression Templates". This concept leads to an optimal performance even on special architectures like vector machines. Furthermore, concepts for storing expressions and code optimizing are presented. In order to verify the usability of these programming techniques in real applications, we discuss a template library which calculates local stiffness matrices arising from Finite Element discretizations.

1 Introduction to expression templates

Expression Templates (ET) were invented independently by Veldhuizen [12] and Vandevoorde [15]. The primary aim of this technique is to provide an efficient C++ implementation of elementary operators like $+$, $-$, for long vectors and other objects. The difficulty is that a simple operator overloading of such operators leads to cache inefficient codes by allocating unnecessary auxiliary vectors and iteration loops for large expressions. To avoid this problem, ET implementations apply operator overloading to build up a type-dependent expression tree which represents all terms of the expression. The evaluation of such an expression tree is performed in the assignment operator by inline functions of the expression tree. By inlining and template concepts in C++, the compiled Expression Template code is as efficient as their corresponding C counterparts.

Expression Templates are used in several applications within the scientific computing community. For instance, applications of Expression Templates in physics are described in [3]. Blitz++ [14] and uBLAS [16] are two linear algebra libraries which use ET and reach the performance of C or FORTRAN implementations. PETE (Portable Expression Templates Engine) was introduced to support programmers in coding the ET technique and offered ways to use ET on STL containers [4]. POOMA [8] supports users in implementing mathematical algorithms associated with Finite Difference discretizations of PDEs. Finally, in [10], it was shown that ET are very helpful for implementing a Finite Element library with a user friendly interface.

However, many C++-programmers still hesitate to implement the ET technique. One reason is that ET is an unusual implementation technique for most C++-programmers. But there is also a fundamental drawback of the classical Expression Template technique. To explain this, let us consider the following implementation of the `operator+` (addition of two expressions) of the ET implementation referring to [15] (see Listing 1).

Communicated by G. Wittum.

J. Härdtlein · C. Pflaum (✉)
Department of Computer Science 10, University of Erlangen-Nuremberg, Cauerstr. 6, 91058 Erlangen, Germany
e-mail: pflaum@informatik.uni-erlangen.de

A. Linke
Weierstrass Institute for Applied Analysis and Stochastics, Mohrenstr. 39, 10117 Berlin, Germany

C. H. Wolters
Institute for Biomagnetism and Biosignalanalysis, University of Muenster, 48149 Muenster, Germany

Listing 1 Part of a traditional expression templates implementation

```
template < class A, class B >
Expr < Binary_Expr < Expr < A >,Expr < B >,Plus > >
 operator + (const Expr < A > & a, const Expr < B > & b) {
     typedef Binary_Expr < Expr < A >,Expr < B >,
         Plus > Type;
     return Expr < Type > (Type(a,b));
   }
```

This overloading of the `operator+` can only be applied to add two general expressions. However, additional overloadings should be implemented for sum of an expression and a vector, a vector and an expression and addition of two vectors. This means that at least four overloaded versions have to be implemented for every binary operator. Even more variants of operators are needed when the operands include constant fundamental data types (e.g. double values).

In the next section, we introduce a simplified mechanism for implementing ET, decreasing the necessary coding up to 55% of traditional codes. Section 3 presents enhancements of the ET technique, including Fast ET and automatic return type generation. To demonstrate the efficiency of these ET advancements, we discuss *Colsamm*, a C++ template library for calculating local stiffness matrices arising from FEM-discretized PDE operators. Additionally, we outline some applications of *Colsamm*.

2 Easy expression templates

In this section, we present a new technique for implementing ET, which simplifies the application of ET in numerical libraries. The main idea of this new technique is the use of the *curiously recurring template pattern (CRTP)* mechanism in C++. Before explaining the application of this mechanism in detail, let us describe the concept of a wrapper class in an expression template implementation.

2.1 Wrapper class and inheritance at compile time

The complexity of common expression template implementations results from the template interfaces of the multiple overloaded operators. However, the operator overloading in C++ works on general template types as well (e.g. ...**operator+** (**const** A& a, **const** B& b)...). Therefore, general overloaded versions of one operator often lead to ambiguous operator versions, especially when mixing expression template implementations. Thus, a type-secure operator overloading with ET requires a common interface which encapsulates all occurring expressions. This common interface is implemented by a wrapper class, which handles an object representing an expression and refers all function calls to this stored expression object. Generally, this wrapper class serves as an appropriate interface and can be equipped with further abilities.

The consequence of a wrapper class is that one has to implement the `operator+` several times as explained in the introduction. To avoid this, we apply the CRTP technique [13] for the operation classes (e.g. addition or multiplication) and the underlying data structures (e.g. vectors or monomials). For example, let X be an operation class, i.e. `class sum`, or a class representing a vector, i.e. `class Vector`. Then, this class X inherits from the templated wrapper class using X itself as template argument. Then, any wrapped object can be converted into its encapsulated object and the corresponding type is determined at compile time using the template argument of the object. In order to perform these conversions automatically, we implement a cast operator in the wrapper class, which yields an object of the derived class type.

Listing 2 shows a wrapper class (Expr) using the CRTP technique. Observe that the cast operator in Listing 2 returns a constant reference (**const** E&). The internal conversion is applied to pointers (∗**static_cast**<**const** E∗>(**this**)). Hence, a recursive calling of the operator itself is avoided. This would have occurred if we had used references internally.

Listing 2 Implementation of CRTP-based wrapper class

```
template <class E >
struct Expr {
  operator const E & () const {return *static_cast<const
    E* >(this);}
};
```

2.2 Easy expression templates

Using the class Expr as in Listing 2, we obtain a simple and type-secure expression template implementation. This is shown in Listing 3 for a component-wise sum of vectors.

Listing 3 Easy expression templates concerning sums of expressions

```
template < class L, class R >
class Sum : public Expr < Sum < L,R > > {
     const L & l; const R & r;
   public:
5    Sum(const L & l_, const R & r_) : l(l_), r(r_) {}
     double give(int i) const { return l.give(i)+r.
     give(i); }
 };
template < class L, class R >
10 inline Sum < L,R >
 operator+ (const Expr < L > & l, const Expr < R > & r) {
      return Sum < L,R > (l,r);
 }
class Vector : public Expr < Vector > {
15    int size;
      double *data;
   public: // Constructors, destructor ...
```

```
    double give (int i) const { return data[i]; }
    template < class E >
20  void operator= (const Expr< E > & e_) {
      const E & e( e_ );
      for (int i=0; i<size; ++i)
        data[i] = e.give(i);
      }
25  };
```

The first part of Listing 3 shows the class Sum, which represents the sum of two expressions. This class is derived from the wrapper class using CRTP. The constructor initializes the constant data member references, while the function give(**int** i) performs addition of the i-th component and refers further evaluations to the stored operands. Additionally, lines 9 to 13 contain the overloaded plus operator, which returns an operating object. Since casting of the wrapped objects to its derived object is automatically applied, the parameters of the operators fit to the operating class constructor (see line 12). Note that the vector class is derived from the wrapper class as well. Since a Vector class object can be converted to Expr<Vector>, no further overloading of the plus operator is required. In Listing 3, we skip the constructors and the destructor of the Vec\−tor class and outline the give(**int** i) function, which returns the i-th entry of the vector data array. Observe that the overloaded assignment operator requires an argument of type Expr . In the assignment operator, evaluation of the expression is performed by the function give(**int** i). To this end, an explicit cast of the wrapped expression to the derived object is needed (see line 20).

The ET version explained above is more compact and shorter. Therefore, it is easier to understand and to implement compared to the usual codings. Furthermore, today's compiler lead to the same performance for both ET implementations. Moreover, we observed that older compilers achieve more efficient codes in case of Easy ET.

Based on the above implementation, vector-like expressions can be implemented in the same way as their mathematical notation (see Listing 4).

Listing 4 Usage of the ET library

```
int main() {
    Vector a, b, c;
    ...
    c = a + b * a;
}
```

This Easy ET version facilitates programmers to realize libraries with even more complex data structures than vectors. For instance, expression templates can be applied to describe polynomials, differential operators, and other mathematical objects (see Sect 4).

3 Expression templates advancements

In this section, we explain several advanced expression template concepts. The main aim of these concepts is to improve the efficiency of ET. But, we also present concepts for storing expressions, such that an expression can be used in different parts of a code.

3.1 Expression templates performance studies

The most detailed study concerning performance problems of ET implementations were reported by Bassetti, Davis and Quinlan [1]. They discovered the effect of register spillage, which is basically caused by leaks in matching the aliases of the vector's data pointers during the evaluation of ET expressions.

In order to motivate the enhancements concerning the ET implementations presented below, let us consider the expression

$$c = a + b * a.$$

Here, the sum and the multiplication are applied component-wise. Then, the Easy ET implementation in Section 2 yields the following implementation of the assignment operator (after instantiating the template parameters):

```
c.operator= (const Expr<Add<Vector,Mult<Vector,
Vector > > & e_){
  const Add<Vector,Mult<Vector,Vector> & e (e_);
  for (int i=0; i<size; ++i)
      data[i] = e.give(i);
}
```

By inlining the give(i) functions, the right-hand side of the component-wise evaluation is expanded to

```
c.data[i] = e.give(i);
          = e.l.give(i) + e.r.give(i);
          = e.l.give(i) + e.r.l.give(i) * e.r.r.give(i);
```

In this expression, the compiler has to substitute the give(i) calls by an array access to the underlying arrays, a.data[i] and b.data[i]. But, due to the conservative restrictions in the aliasing concept, the compiler cannot ensure that the first and the third variable point to the same unchanged data of a.data[i]. Since several loads are started, this aliasing leak creates performance problems, even if the data is already resident in cache. Such optimization problems mainly affect ET implementations on high performance platforms, e.g. vector or shared memory platforms.

To support compilers in optimizing ET programs, we introduced the Fast ET technique, which was first published in restricted form in [9]. In [5] this method was studied in a more implementable manner. It was observed that Fast

ET lead to performance improvements even in case of usual workstations with standard compilers. In this paper, we present a different implementation technique for Fast ET. This implementation technique is much easier to implement and to understand than the old versions in [9] and [5].

3.2 Fast expression templates

An interesting property of ET is that the whole tree structure of the expression is contained in the type of the expression template. The idea of Fast ET is to use this type for evaluation of an expression. To this end, the member function give(i) is defined to be a static function. Furthermore, all data have to be defined as static data. This implies that two vectors or two constants can only be distinguished by the type of the object. This is obtained by introducing a template parameter which enumerates objects like vectors or constants. This concept can be viewed as meta programming. Listing 5 shows the Fast ET counterpart of the program of Listing 3.

Listing 5 Fast expression templates implementation

```
template<class E>
   struct FastExpr {};
template<class L, class R >
struct FastSum : public FastExpr< FastSum < L,R > >  {
  static double give(int i) { return L::give(i) +
    R::give(i); }
 };
template<class L, class R>
FastSum < L,R >
  operator+ (const FastExpr < L > & l,const
     FastExpr< R > & r){
     return FastSum<L,R>();
 }
template<int id>
class FVector : public FastExpr< FVector<id> > {
      int size;
      static double* data;
   public: // Constructors, destructor ...
      static double give(int i) { return data[i]; }
      template <class E>
      void operator= ( const FastExpr< E > & e_ ) {
          for (int i=0; i < size; ++i)
             data[i] = E::give(i);
        }
 };
template<int id> double* FVector<id>::data = NULL;
```

This complete meta version of ET has to be applied in a slightly different way compared to traditional ET implementations. Since Fast ET have static data pointers, all vectors have to be enumerated by an unique template integer. The example of Listing 4 has to be implemented as follows:

Listing 6 Usage of the fast ET library

```
   FVector < 1 > a; FVector < 2 > b; FVector < 3 > c;
   ...
   c = a + b * a;
```

Fast ET require similar implementations for all constant values like 2.0 or 4.7.

Focusing on the performance of this enhanced ET technique, we compared some Fast ET implementations (FET) with their classical ET (CET) and C counterparts (NET), respectively. Our studies were performed on high performance platforms as well as on workstations. First, we recognized, that the enhanced meta ET implementations compiled up to ten times faster than the classical codings. Nevertheless, the compile time of every ET implementation increases polynomial with respect to the depth of the expression tree.

Concerning the performance of the executables, we restrict ourselves to two graphs resulted from the NEC SX-6/48M6 at the HLRS Stuttgart, Germany, which is a shared memory vector system [11]. The samples were executed on one node (covering 8 CPUs) with automatic vectorization. The behavior of the performance graphs is similar on further tested platforms, e.g. the Hitachi SR 8000 supercomputer (LRZ Munich) or an AMD Opteron Cluster using OpenMP.

Figure 1 depicts the execution time per iteration on NEC SX-6 with respect to the vector size. The graphs show the computations of a 2D Poisson (nine-point stencil) and 3D Poisson (21-point stencil) problem, respectively. In order to achieve an easy vectorizable solver we implemented the Jacobian-method. The performance results show that Fast ET lead to the same performance as its C counterpart.

Listing 7 Vectorizable Jacobian solver of the 2D poisson problem

```
 FVector < 0 > x(size);    FVector < 1 > dummy (size);
 FVector < 2 > rhs(size);  FVector < 3 > exact (size);
// Assign rhs and exact_solution  ...
 Const < 1 > const1 = 3./8.; Const < 2 > const2 = 1./8.;
 while ( eps < L2_Norm (exact-x) ){
    dummy = const1 * rhs + const2 *
          (N(x) + NW(x) + W(x) + SW(x) + S(x) + SE(x)
                + E(x) + NE(x));
    x = dummy;
 }
```

In addition, Listing 7 displays some excerpts of the coding used for the 2D problem focusing on the implementation of the Jacobian solution step. Herein, N,NW,... denote the directions north, north-west, and so on, respectively. This example demonstrates the usage of a suitable class, that encapsulates the constant values within an Fast ET expression.

3.3 Fast expression templates in practice

When using Fast ET in real applications, only slight differences arise in comparison to classical ET codings. The code fragment in Listing 7 demonstrates how the vector and scalar variables have to be enumerated. Generally, the user is responsible for the unique enumeration of the vectors and their application. However, there are ways to support an user

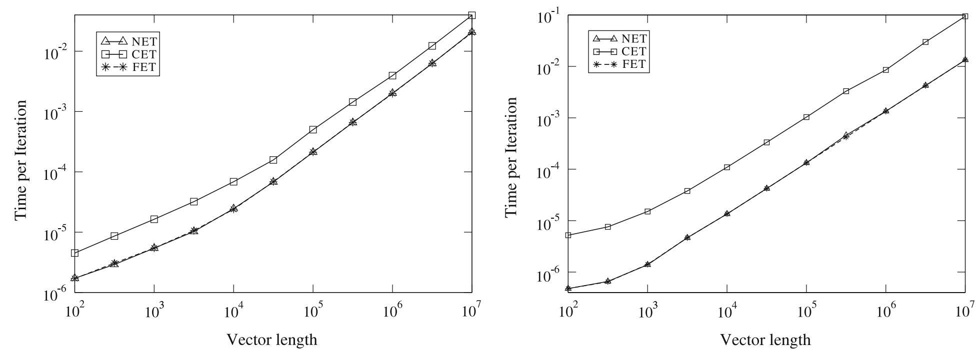

Fig. 1 Performance results on the NEC SX-6 vector computer for Poisson's problem in 2D and 3D, respectively. The figures show the execution time per iteration by increasing vector size, displayed in logarithmic scale

by implementing checks for unique initialization and suitable warnings.

Furthermore, the Fast ET technique has to be used only for the performance critical parts. For example, the initialization of the vectors and the computations that do not require intensive efforts can still be handled by a non-static ET implementation. In more detail, at the beginning of a Fast ET block, the enumerated vectors have to be initialized by assigning the non-static vector arrays to the static data pointers. We emphasize that this initialization need not be a deep copy. Instead, only the pointer addresses have to be assigned. Afterwards, the performance critical calculations can be performed, using the Fast ET library. Finally, after finishing the computations, no data pointer reassignment is needed, since the former variables still point to the corresponding data arrays.

3.4 Return type minimization

This subsection addresses some implementation challenges arising from ET and operator overloading. Even if this technique is already used in some ET implementations, we present this mechanism to show the flexibility of this approach. To this end, we explain this approach for the Easy ET implementation in Listing 3.

The accessing functions give(**int** i), presented in the previously listings, have a specific return type. While programming an ET library, we would like to get away from fixing the base type of our vector class. Moreover, in some applications it is important to have versions of vectors relying on, e.g. double and complex<double> base data type and it is necessary to combine different types of variables within expressions. The performance would be improved significantly, if the evaluation of the expression is performed with the actual smallest data type (e.g. double) in all parts of the expression.

Actually, this type information is already known at compile time, and thus we recall a mechanism that is already used in PETE, see [4]. The partial specialization of template types, i.e. traits, is used to specify the optimal expression-dependent return type of a member function. Listing 8 shows an implementation of a return type minimization according Table 1 for the types **double** and complex<double>.

Listing 8 Traits accomplishing the return type minimization

```
template < typename L, typename R >
 struct ResultType {
   typedef std::complex < double > Type;
 };
template <>
 struct ResultType <double,double > {
   typedef double Type;
 };
```

The return type minimization **struct** ResultType is used in the ET implementation in Listing 9. Observe the typedef definition in the public part of the operating classes and the Vector class. This typedef is used to calculate the minimal return type of the operation. Obviously, this return type minimization via traits is extendable to more than two types. Then, more types have to be implemented via additional specializations.

The return type minimization allows to construct ET implementations, that are independent from an underlying data type. This means that the same ET implementation can be used for objects of different type. This is helpful for constructions like Vector<Vector<double> > in an ET library.

Listing 9 Usage of return type minimization in the easy ET implementation

```
template <class L, class R>
 class Sum : public Expr< Sum < L,R > > { ...
  typedef typename ResultType < typename L::Type,
                            typename R::Type >::Type Type;
   Type give(int i) const { return l.give(i) + r.give(i); }
 };

template < typename baseType >
 class Vector : public Expr< Vector<baseType> > { ...
     baseType* data;
   public:
     typedef baseType Type;
     // Constructors, destructor ...
     Type give(int i) const { return data[i]; }
 };
```

Table 1 Possible type combinations of binary expressions basing on dou−ble and com−plex<dou−ble> types

Type of expression L	Type of expression R	Resulting return type
double	double	double
double	complex<double>	complex<double>
complex<double>	double	complex<double>
complex<double>	complex<double>	complex<double>

3.5 Expression-dependent specializations

An ET implementation provides mechanisms for handling and evaluating any expression combinations, which might be built via the overloaded operators. However, on high performance platforms, special cases may occur, where some handcrafted implementations still perform significantly faster than the ET codings. This might be a very efficient implementation of the matrix-vector multiplication, an optimized Gauss-Seidel iteration, or an optimized code on a special hardware. In this case, one can implement an additional overloading of the assignment operator for that kind of expression. Then, the compiler chooses this specialized implementation of the assignment operator instead of the general template implementation. As a consequence, the optimal implementation of an expression is automatically chosen by the compiler.

Within *Colsamm*, we use the template specialization in order to handle gradient vectors arising from the first derivatives of basis functions in a different way than arbitrary vectors containing problem-specific spatial data (see Section 4.1).

3.6 Mechanisms for reusing ET objects

Flexible and user-friendly ET libraries require mechanisms for storing and reusing expressions. This can be obtained by virtual functions or by type encapsulation for codings based on Fast ET.

Let us first explain the concept of using virtual functions. Concerning the Easy ET implementations in Listing 3, the coding has to be modified such that all member variables of the ET-operating classes are no longer references but full variables. To avoid unintended copies of data arrays of vector object, we introduce a template specialization of the wrapper class which handles the vector objects as references (end of Listing 10). Then, we add an abstract base class on top of the wrapper class that provides the virtual functions giveV(**int** i). These are implemented in the wrapper class and refer to he corresponding functions of the underlying operating classes. The virtual functions are named differently (giveV instead of give), in order to facilitate full inlining of the ET function calls. Finally, Listing 10 sketches a storable version of the previously presented ET codings.

Listing 10 Code fragment concerning the storage of ET objects

```
struct Base {
  virtual double giveV (int i) const = 0;
  virtual ~Base(){};
};
// the wrapper class
template <class E>
 struct Expr : public Base {
  operator const E& () const {return *static_cast<const
   E*>(this);}
  virtual double giveV (int i) const {
     return ((const E& )(*this)).give(i);
  }
};
template <class L, class R >
 class Sum : public Expr < Sum < L,R > > {
   const L l; const R r;
  public:
   Sum (const L& l_, const R& r_) : l(l_), r(r_) {}
   double give (int i) const ; // as before
 };
// special wrapper for vectors
template <class E>
 struct Expr <const E& > : public Base {
  operator const E& () const {return *static_cast<const
   E*>(this);}
  virtual double giveV (int i) const {
     return ((const E& )(*this)).give(i);
   }
 };
class Vector : public Expr<const Vector& > {
     ...
   void operator= (const Base* expr){
       for(int i=0; i< size; ++i)
         data[i] = expr->giveV(i);
    }
 };
```

By the above construction, we can store every expression by assigning it to a Base* pointer. However, this implementation works with deep copies of the occurring ET objects and can decrease the performance while building the expression. Indeed, virtual function calls cannot be inlined. But the nested calls of the underlying ET operating classes are still optimized in the expected manner. Consequently, this mechanism for storing ET is applicable with a slightly

reduced efficiency when it is used in performance critical parts. An application of this concept of storing expressions is explained in Sect. 4.1.3. In this application, finite element basis function are described by expressions and the corresponding expression template is stored using the above concept.

In performance critical parts one should not apply the concept of storing ET with virtual functions. In this case, we recommend Fast ET and storing expressions by type encapsulation. Then, there is no performance loss.

Using a typedef, an expression template based in Fast ET can be stored by assigning its type to a new type name. Since Fast ET work exclusively on types, this new type name contains all informations about the expression. The new type name is evaluated using the static functions of this new type. Since Fast ET implementations operate on static data within Vec\–tor and Const classes, the evaluation of the new type name leads to the same performance as the evaluation of the original Fast ET (see application in Sect. 4.1.2).

Listing 11 Code fragment concerning the storage of ET objects

```
FVector < 1 > a; FVector < 2 > b; FVector < 3 > c;
...
typedef __typeof__(a + b*a) Expression1;
...
// initialize Vectors a, b
 c = Expression1();
```

4 Expression templates in action

In this section, we explain how to apply the different ET concepts of the previous section. To this end, we focus on the numerical solution of PDEs. Obviously, ET can be used in the linear algebra part of a PDE solver. Here, we explain how to apply different ET concepts for calculating the local stiffness matrices of a FE-discretization of a PDE. For demonstrating the practicability of ET, we implemented the concepts in a library *Colsamm* (see [6]).

Colsamm uses different advanced Expression Templates techniques:

- In Sect 4.1.2, mapping functions from the reference element to an arbitrary element are implemented by Fast ET, since optimal performance is required. These ET are stored and repeatedly reused (see Sect. 3.6).
- In Sect. 4.1.3, basis functions of the finite element space are described with Easy ET based on virtual functions according to Sect. 3.6. Here, performance issues do not exist, but the expression objects have to be reused.
- In Sect. 4.1.4, the weak form of a partial differential equation is described by Easy ET.

4.1 Colsamm - Computing local Stiffness Matrices

4.1.1 Basic goals and ET applications

The FEM is an important discretization technique for partial differential equations. To implement this method, it is necessary to calculate local stiffness matrices. The aim of the library *Colsamm* is to provide a helpful tool for the calculation of these local stiffness matrices.

To explain the interface of *Colsamm*, let us recall the basic definition of a Finite Element according [2]. A Finite Element is a triple (K, P_K, S_K), where

- K represents the geometry of the element, e.g. triangle, hexahedron, etc.,
- P_K defines a space of functions on K, usually polynomials, and,
- Σ is the set of linear functionals on P_K, which represent the element-specific *degrees of freedom*.

To calculate the local stiffness matrix of an element, we additionally need a mapping function

$$F_K : \hat{K} \to K,$$

where $\hat{K}$ is a fixed reference element. This mapping induces a space of functions $P_{\hat{K}}$ on the reference element $\hat{K}$. Furthermore, let $a_K : P_K \times P_K \to \mathbb{R}$ (or $\mathbb{C}$ be the bilinear form corresponding to the variational formulation of the PDE. Let us assume that this bilinear form is given in an integral form such that the integrand I is known. *Colsamm* contains different expression interfaces for the description of the following three mathematical objects:

- the set of basis functions of $P_{\hat{K}}$ corresponding to Σ,
- the mapping $F_K : \hat{K} \to K$, and
- the integrand I of the bilinear form a_K within the variational formulation of the problem.

In the following sections we sketch the ET implementations within *Colsamm*. Moreover, we compare the interface of the library with corresponding mathematical notations. Further examples and applications of *Colsamm* can be found in [6].

4.1.2 Mapping definition via fast ET

The implementation of the mapping function is a performance critical part, since the mapping function is needed for the evaluation of the integrand at the Gaussian points on the reference element. Therefore, *Colsamm* provides an interface for describing the mapping function $F_K : \hat{K} \to K$ by using the Fast ET technique. To explain this interface, let us restrict to the case of a tetrahedron and a linear mapping. A linear mapping function can be described as follows:

$$T(u,v,w) = P_0 + (P_1 - P_0)u + (P_2 - P_0)v$$
$$+(P_3 - P_0)w \quad .$$

Here P_0, P_1, P_2, P_3 denote the vertices of the tetrahedron K, and u, v, w represent the three coordinates on the reference element $\hat{K}$.

Corresponding to the points P_i, *Colsamm* defines the objects P_0(), P_1(), P_2(),P_3().

By applying the macro

```
#define Define_Element_Transformation(A) typedef __
typeof__(A)
```

the user of *Colsamm* can store the corresponding Fast ET into the type called Tetrahedron_Transformation as follows:

```
Define_Element_Transformation (
    P_0() + ( P_1() − P_0()) ∗ _U() +
            ( P_2() − P_0()) ∗ _V() +
            ( P_3() − P_0()) ∗ _W() )
  Tetrahedron_Transformation;
```

With the help of the above macro a new type is defined, which later can be used for further calculations inside the library *Colsamm*.

4.1.3 User-defined basis functions

Let $(b_i)_{i\in\Sigma}$ be the set of nodal basis functions of a finite element. One can show, that these basis functions have to be evaluated only one time at each Gaussian point of the reference element independent of the number of finite elements. Thus, the description of the nodal basis functions is not a performance critical part of the code. Furthermore, there exist applications (e.g. interpolation at certain points), which require to save the basis functions in order to enable additional evaluations after the initialization of the basis functions. Therefore, *Colsamm* applies the concept of Easy ET with virtual functions for describing and storing the expression of each basis function b_i (see Sect. 3.6). For the definition of the basis functions, the library provides monomials, trigonometrical and exponential functions. Additionally, it is possible to introduce vectorial basis functions.

As an example, let us consider linear finite elements on a tetrahedral element. The corresponding nodal basis functions on the reference element are:

$$\lambda_0(x,y,z) = 1 - x - y - z, \quad \lambda_1(x,y,z) = x,$$
$$\lambda_2(x,y,z) = y, \quad \lambda_3(x,y,z) = z.$$

User-defined finite elements in *Colsamm* are introduced by defining special element classes, that derive from a general element class with appropriate template parameters. Concerning the definition of a tetrahedron, note that

- it has four corners,
- we intend to introduce four basis functions,
- a tetrahedron is a three-dimensional polygon,
- the reference element is the unit tetrahedron, and
- the unit element can be mapped to a general tetrahedron by applying the previously defined affine linear mapping Tetrahedron_Transformation.

For defining a tetrahedral element, we put these above data into template parameters of the class _Simple_Element_ and derive the class _Tetrahedron_. Additionally, the basis functions have to be implemented in the corresponding constructor. Listing 4.1.3 shows the complete definition of the tetrahedral element using *Colsamm*.

```
struct _Tetrahedron_
 : public _Simple_Element_<4,4,D3,Unit_Tetrahedron,
                                  Tetrahedron_Transformation> {
   _Tetrahedron_() {
      this->Set ( 1. − X_() − Y_() − Z_() ) ;
      this->Set ( X_()  ) ;
      this->Set ( Y_()  ) ;
      this->Set ( Z_()  ) ;
   }
} ;
```

More complex elements can be built by changing the number of basis functions and the template parameters. *Colsamm* already provides linear finite elements on elementary geometries K. Furthermore, the library includes special constructions for vector finite elements.

4.1.4 Implementing the integrand

Let us assume that the integrand of the bilinear form $a(v, w)$ is given. *Colsamm* provides an interfaces to describe the expression of this integrand using the Easy ET technique. This interface includes differential operators like grad and d_dx. The functions v, w are abbreviated by v_() and w_() . The member function integrate of the user-defined finite element returns the local stiffness matrix for a given finite element.

Let us explain this in more detail in case of Poisson's problem using linear elements on tetrahedra. The local bilinear form of this problem is

$$a_K(v,w) = \int_K \nabla v \cdot \nabla w \, \mathbf{d}\mu,$$

where K is a tetrahedron. Then, the corresponding local stiffness matrix is a 4×4 matrix. Using *Colsamm*, this local stiffness matrix can be calculated as follows:

```
_Tetrahedron_ myTetrahedron;
 std::vector<std::vector<double> > local_matrix;
// loop over the elements
// pass the coordinates of the actual element's vertices
```

```
  myTetrahedron(actual_vertices);
  ...
// computation of the local stiffness matrix
  local_matrix = myTetrahedron.integrate(grad(v_()) *
   grad(w_()));
```

As a second example, let us consider the variational formulation of Maxwell's equations. The corresponding bilinear form is:

$$a_K(v, w) = \int_K \nabla \times v \cdot \nabla \times w + \varepsilon_K(x, y, z) v w \, \mathbf{d}\mu.$$

This bilinear form can be described in a natural way by *Colsamm*:

```
  local_matrix = myTetrahedron.integrate(
            curl(v_vec3D()) * curl(w_vec3D()) +
                   epsilon*v_vec3D()*w_vec3D());
```

where, epsilon can either be a constant value or a variable function.

4.2 Applications of *Colsamm*

The library *Colsamm* is used for calculating local stiffness matrices in different applications like laser simulation, simulation of optical flows, simulation of melting furnaces, and computations for dipole modeling. Let us explain two of these applications in more detail:

Electroencephalography (EEG) dipole source analysis In [17], *Colsamm* was applied to Electroencephalography (EEG) dipole source analysis. In this application, software developers in Münster, Germany, used Colsamm as a module. However, the developers of Colsamm work in Erlangen, Germany. Usually, the collaboration of different research groups on a common software project is not an easy task. Here, expression template were very helpful for describing a clear interface between the library *Colsamm* and the application
code.

Now, we briefly describe the application itself. In EEG dipole source analysis, it is necessary to solve a Poisson-type equation with different anisotropies and a dipole source on the right hand side. Colsamm assembles the local stiffness matrices and the right hand side of the problem. By user-friendly interface of *Colsamm*, this only requires a few lines of code. Figure 2 shows a numerical result obtained by the EEG dipole source analysis using the Colsamm module.

Laser simulation In laser simulation, different kind of physical effects have to be simulated. These effects range from thermal lensing effects to optical effects caused by the construction of the laser resonator. For the simulation of these effects, we use a finite element library based on expression templates. This finite element library *BlockUG* provides differential operators on block structured grids using trilinear elements. These differential operators can be defined by the user of the library. The application of expression templates allows a user friendly description of differential equations. For the calculation of the local stiffness matrices *BlockUG* calls *Colsamm*. Therefore, the expression template library *BlockUG* applies *Colsamm* as a module.

These examples show that expression templates are very useful for the development of user friendly libraries. Nevertheless, we would like to mention that there exist other projects which provide highly efficient libraries for the calculation of local stiffness matrices (see [7]).

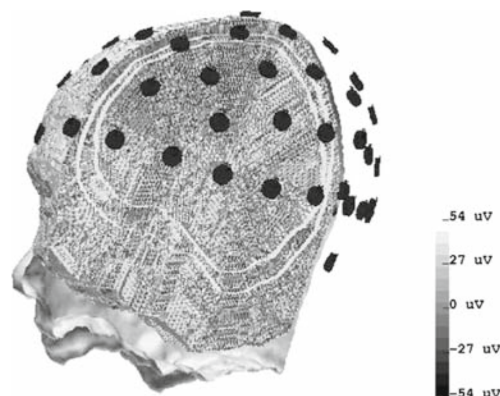

Fig. 2 *Left* Sliced tetrahedral finite element head model with EEG (Electroencephalography) electrodes (international 10/20 system). The surfaces of inner and outer skull are indicated in the figure. *Right* Electric potential distribution of a quasi-radial dipole source in the primary somatosensory cortex (area SI) computed in a three compartment (skin, skull, brain) realistically-shaped geometry-adapted hexahedra finite element head model with an anisotropic skull compartment

5 Conclusions and perspectives

In this paper, we reviewed different advanced ET techniques, which could spark new interest in this C++ programming technique. Generally, ET libraries enable readable application code on a high level of abstraction combined with high performance of the executed code. But, programming ET libraries is often viewed as difficult, and optimal performance is not always assured. The presented ET techniques lead to codes of optimal performance and allow an easier and more reliable coding of ET libraries.

The concept of Easy ET, presented in Sect. 2, improves classical ET by means of readability and maintainability (see [13]). Also, the compile-time of Easy ET implementations is sometimes reduced in comparison to classical implementations.

The Fast ET technique, presented in Sect. 3, is a technique to avoid some performance problems of classical and Easy ET implementations. Here, the use of Fast ET implementations yields C++ code with optimal performance in comparison to C implementations. Since Fast ET need a quite crude handling by the programmer, we recommend this technique only for the performance relevant parts of a software, when optimal performance is not obtained by Easy ET. However, by the optimization facilities of modern C++ compilers, this is very seldom the case. Nevertheless, Fast ET make clear where performance problems of ET implementations may result from.

Further, we introduced some ET related programming techniques, which are useful in scientific computing. "Return Type Minimization" assures optimal performances when different types are used together in a single expression, e.g. double and complex. "Expression-dependent Specializations" allow to use optimized software code for special expressions like a matrix-vector multiplication, without loosing the readability of ET application code. "Mechanisms for Reusing ET Objects" allow to save ET Objects and to reuse them later.

Last but not least, we introduced the FEM library *Colsamm*, which extensively uses the proposed advanced ET techniques. *Colsamm* allows user-friendly and highly performing computation of local stiffness matrices in finite element discretizations of partial differential equations. It demonstrates that ET techniques in scientific computing are not limited to linear algebra computations, but can support the development of complex, reliable and efficient mathematical libraries on a much higher level of mathematical abstraction. Therefore, *Colsamm* is used with great success for teaching the finite element method in a Bachelor program of Computational Engineering at the University of Erlangen-Nuremberg.

References

1. Basetti, F., Davis, K., Quinlan, D.: C++ Expression Templates Performance Issues in Scientific Computing. CRPC-TR97705-S (1997)
2. Ciarlet, P.G.: The Finite Element Method for Elliptic Problems. SIAM (2002)
3. Haney, S.: Beating the abstraction penalty in C++ using expression templates. In: Dubois, P. (ed.) Computers in Physics, vol 10, No. 6, pp. 552–557. American Institute Of Physics, USA (1996)
4. Haney, S., Crotinger, J., Karmesin, S., Smith, S.: PETE: the portable expression templates engine. Dr. Dobb's J. Softw. Tools, **24**(10):88, pp. 90–92, 94–95 (1998)
5. Härdtlein, J., Linke, A., Pflaum, C.: Fast expression templates. In: Suneram, V.S., Albada, G.D.V., Sloot, P.M.A., Dongarra, J.J. (eds.), Computational Science—ICCS 2005, volume 3515 of LNCS. Springer, ISBN-10 3-540-26043-9, ISBN-13 978-3-540-26043-1, ISSN 03-2-9743. pp. 1055–1063 (2005)
6. Härdtlein, J., Pflaum, C.: Efficient and user-friendly computation of local stiffness matrices. In: Hülsemann, F., Kowarschik, M., Rüde, U. (eds.), 18th Symposium Simulationstechnique ASIM 2005 Proceedings, volume 15 of Frontiers in Simulation. ASIM, SCS Publishing House, ISBN 3-936150-41. pp. 748–753 (2005)
7. Kirby, R.C., Logg, A.: Efficient compilation of a class of variational forms. ACM Trans. Math. Softw. **33**(3), Article 17 (2007)
8. Los Alamos National Laboratories. POOMA. www.acl.lanl.gov/pooma
9. Linke, A., Pflaum, C.: Fast expression templates for the SR 8000 supercomputer. In: Stiegritz, J., Davis K. (eds.) Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC). Technical Report, FZJ-ZAM-IB-2003-09, (2003)
10. Pflaum, C.: Expression templates for partial differential equations. Comput. Visual. Sci. **4**, 1–8 (2001)
11. High Performance Center Stuttgart: The NEC SX-6 Cluster Documentation. http://www.hlrs.de/hw-access/platforms/sx6/user_doc (2005)
12. Veldhuizen, T.: Expression templates. C++ Report **7** (5), 26–31 (1995)
13. Veldhuizen, T.: Techniques for Scientific C++. Indiana University Computer Science Technical Report No 542, Version 0.4 (2000)
14. Veldhuizen, T.: Blitz++ User's Guide—Version 1.2. http://oonumerics.org/blitz/manual/blitz.html. (2001)
15. Vandevoorde, D., Josuttis, N.: C++ Templates—The Complete Guide. Addison-Wesley, Reading (2003)
16. Walter, J., Koch, M.: uBLAS, Boost C++ Libraries - Basic Linear Algebra. http://www.boost.org/libs/numeric/ublas/doc/index.htm. (2002)
17. Wolters, C.H., Köstler, H., Möller, C., Härdtlein, J., Grasedyck, L., Hackbusch, W.: Numerical mathematics of the subtraction method for the modeling of a current dipole in EEG source reconstruction using finite element head models. SIAM J. Sci. Comput **30**(1), 24–45 (2007)