Computing (2012) 94:215–228
DOI 10.1007/s00607-011-0176-6

Interval arithmetic using expression templates, template meta programming and the upcoming C++ standard

Marco Nehmeier

Received: 25 March 2011 / Accepted: 14 November 2011 / Published online: 29 November 2011

Abstract In this paper we will discuss different realizations for an efficient interval arithmetic implementation using expression templates and template meta programming in C++. We will improve the handling of the rounding mode switches using expression templates and show how the constructed expression trees can be combined with other features like automatic differentiation. For a further improvement of the run time performance we try to move as many functionality as possible to the compile time using template meta programming techniques. In addition we will illustrate how an interval arithmetic implementation will profit from new features and keywords defined in the upcoming C++ standard.

Keywords Interval arithmetic · Automatic differentiation · Expression templates · Template meta programming · Compile time code optimization · C++ · C++0x

Mathematics Subject Classification (2000) 65G40 · 65Y04 · 68N15 · 68N19 · 68W30

1 Introduction

Interval arithmetic is set arithmetic working on intervals defined as connected, closed and not necessarily bounded subsets of the reals

$$X = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\} \tag{1}$$

The author has presented the results of this paper during the SCAN 2010 conference in Lyon, September 2010.

M. Nehmeier (✉)
Institute of Computer Science, University of Würzburg, Am Hubland, 97074 Würzburg, Germany
e-mail: nehmeier@informatik.uni-wuerzburg.de

where $\underline{x} = -\infty$ and $\overline{x} = +\infty$ are allowed. The set of all possible intervals together with the empty set is called $\overline{\mathbb{IR}}$. The basic arithmetic operations on intervals are based on powerset operations:

$$X \circ Y = \left[\min_{x \in X, y \in Y} (\nabla(x \circ y)), \max_{x \in X, y \in Y} (\triangle(x \circ y)) \right] \tag{2}$$

On floating point systems the value of the lower bound is rounded toward $-\infty$ (symbol ∇) and the upper bound is rounded toward $+\infty$ (symbol $\triangle$) to include all possible results of the powerset operation on the real numbers. Continuous functions could be defined in a similar manner. Note that monotonicity properties could be used to define the result of an interval operation or function only using the bounds of the input intervals.

2 Interval arithmetic using expression templates

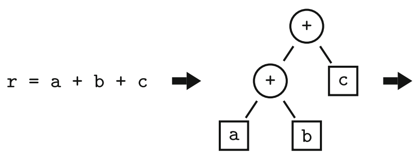
One of the main challenges for an efficient interval arithmetic implementation is the performance penalty for switching the rounding mode to assure the inclusion property of the computation. The problem becomes worse due to poorly implemented floating point functions[1] which require a rounding mode toward the nearest floating point number (symbol $\Diamond$) to work correctly. Hence, interval routines have to restore the primary rounding mode after the computation to ensure a valid behavior for the rest of the program.

C++ interval libraries like Boost [7] or filib++ [18] commonly provide two different approaches to handle the rounding mode. The first and slow one is to reset the rounding mode after each interval operation which leads to needless and at least useless switches of the rounding mode for interval expressions with two or more operations like $a + b + c$. The second approach omits the reset of the rounding mode after each interval operation and shifts the responsibility of saving and resetting the primary rounding mode to the user. But both methods have their disadvantage in speed or usability. To avoid these drawbacks of common C++ interval libraries we, in [20], applied expression templates [22] to efficiently control the rounding mode.

Expression templates is a C++ technique to represent an expression as an expression tree by using C++ templates with the benefit that this tree is explicitly visible at compile time. This offers the possibility to delay the execution of the single operations of an expression and evaluate the whole expression at once. Figure 1 shows the steps of an aspired transformation of an interval expression into basic floating point operations using expression templates and applying the identity

$$\triangle x = -\nabla(-x) \tag{3}$$

[1] E.g.: On a Debian Linux system with GCC 4.3.2 and GNU C Library 2.7 the result for the computation of $e^{3.0}$ with rounding toward $+\infty$ is 0.110028.



```
store_rounding();
set_rounding_downward();
t1 = a1 + b1;
t2 = -a2 - b2;
r1 = t1 + c1;
r2 = -(t2 - c2);
reset_rounding();
```

Fig. 1 Aspired transformation of an interval expression into basic floating point operations

to reduce the switches of the rounding mode. Thereby, the variables with the suffix `1` or `2` of the code snippet in Fig. 1 are the lower or upper bounds of the corresponding variables of the expression, respectively.[2]

Applying formula (3) to compute the upper bound of an interval expression by setting the rounding mode toward $-\infty$ and resetting it after the computation only works for the basic arithmetic operations. For correctly rounded standard functions, as those provided by the C++ library CRlibm [9], generally a differing rounding mode is necessary. To take this limitation into account, we, in [20], decided to build an expression tree at compile time and implemented a visitor-object passed through the tree at run time to manage the rounding mode switches.

In spite of this additional object at run time we achieved a measurable performance increase compared to Boost and filib++ for interval expressions with more than one or two[3] operations [20], respectively. Figure 2 shows a comparison of the average run time for 100,000 repetitions of interval expressions with different length.

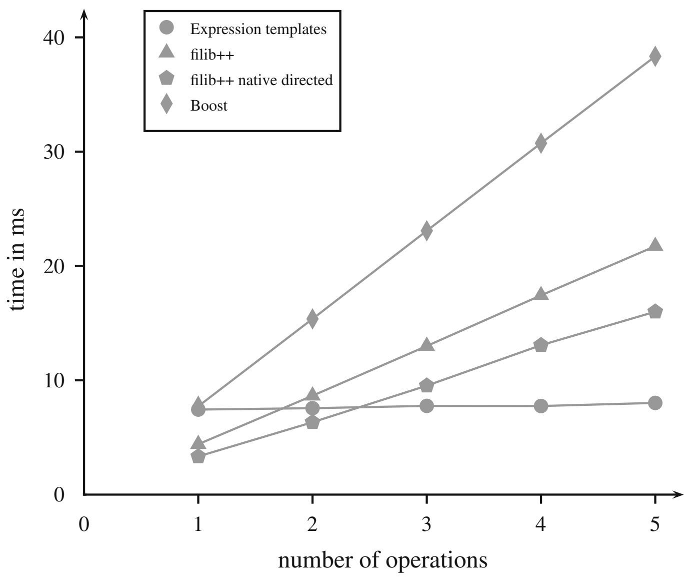


Fig. 2 Performance comparison for interval expressions with different length

[2] Note that the upper bound `t2` of the intermediate result is stored negated.

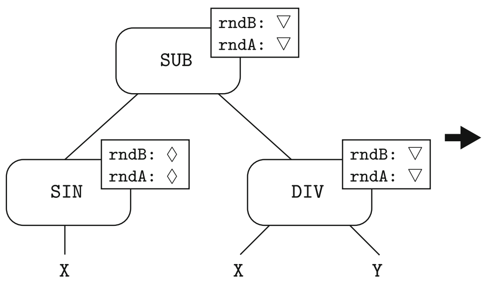
[3] filib++ uses assembler subroutines other than the expression template implementation which uses plain C++. Hence filib++ offers a better performance for short expressions.

3 Improvement of the expression template approach

Up to this point we only used the expression tree to provide an abstract representation of the interval expression at run time to manage the switches of the rounding mode. Moreover, the advantage of an explicitly visible expression tree at compile time should be used for further optimizations and features by applying template meta programming techniques [23] to move as many functionality as possible to the compile time. Furthermore, we show that an implementation could benefit from new features of the upcoming C++ standard [3], called C++0x.

3.1 Optimize rounding mode switches

One characteristic of all basic interval operations and functions is that they require a specific rounding mode to compute one bound of the result. Additionally a unique rounding mode is activated after the computation of the second bound. This knowledge could be used to annotate each operation or function with two states `rndB` and `rndA` to specify the required rounding mode before and the activated rounding mode after the computation. Although the obvious attribution of `rndB` is a rounding toward $-\infty$ and rounding toward $+\infty$ for `rndA` to correspond to the formula (2)[4] a different mapping of `rndB` and `rndA` is commonly used. For basic arithmetic operations the formula (3) could be applied. Hence, the rounding mode toward $-\infty$ is assigned to `rndB` and `rndA` for the interval addition, subtraction, multiplication and division. On the other hand if we use CRlibm [9] for the standard functions we need a rounding mode to the nearest floating point number for both attributes `rndB` and `rndA`. This is necessary because CRlibm uses the processor with the rounding mode to the nearest floating point number to provide correctly rounded floating point functions with the four different rounding modes to the user[5] [10]. Therefore, these correctly rounded floating point functions are used to compute the elementary interval functions using the well known monotonicity properties. Figure 3 shows the annotated expression tree



```
store_rounding();
set_rounding_nearest();
s1 = ($\exists k \in \mathbb{Z} | 2k\pi - \pi/2 \in$ X) ? -1
       : min(sin_rd(x1), sin_rd(x2));
s2 = ($\exists k \in \mathbb{Z} | 2k\pi + \pi/2 \in$ X) ? -1
       : -max(sin_ru(x1), sin_ru(x2));
set_rounding_downward();
t1 = min((y2 >= 0 ? x1 : x2) / y1,
       (y2 >= 0 ? -x1 : -x2) / -y2);
t2 = -min((y1 >= 0 ? x2 : x1) / -y2,
       (y1 >= 0 ? -x2 : -x1) / y1);
r1 = s1 + t2;
r2 = -(s2 + t1);
reset_rounding();
```

Fig. 3 Expression tree with corresponding rounding modes of interval expression $sin(X) - X/Y$ as well as the aspired pseudo code of the code transformation

[4] 1. Set rounding mode $\nabla \equiv$ `rndB`, 2. compute lower bound, 3. set rounding mode $\Delta \equiv$ `rndA`, 4. compute upper bound.

[5] The function names are suffixed with `_rn`, `_ru`, `_rd`, and `_rz` for the rounding modes to the nearest number, toward $+\infty$, toward $-\infty$ and to zero, respectively [10].

for the interval expression $sin(X) - X/Y$ as well as the aspired pseudo code of the code transformation[6]. In this pseudo code the sin is computed with CRlibm functions using the processor in round to nearest mode. For the division and the subtraction the rounding mode toward $-\infty$ is used to compute the result using the algorithms described in [17].

This tree could be generated by expression templates using the class `Interval`[7] for the leaf nodes and the classes `UExpr<class OP, typename A>` and `BExpr<class OP, typename A, typename B>` for the inner nodes. Here, the template classes `UExpr` and `BExpr` are abstract representations for unary or binary expressions specified by the template parameter `OP` as a policy class [1] working on the arguments of the types `A` and `B`, respectively.

Listing 1 shows the structure of a policy class used for a binary expression. The two rounding mode states `rndB` and `rndA` are defined as static constants of an enumeration type `RndState` with enumerators `DOWNWARD`, `UPWARD`, `NEAREST` and `TOZERO` for the provided rounding modes as well as the additional enumerators `UNKNOWN` and `UNUSED`.

```
struct Sub {
   static const RndState rndB = RndState::DOWNWARD;
   static const RndState rndA = RndState::DOWNWARD;

   static Interval eval(Interval a, Interval b) {
    /* compute a - b */
   }
};
```

Listing 1 Policy class for an interval subtraction.

With these classes it is possible to generate the expression tree as a nested template type at compile time, see Fig. 4. Thereafter a trait class [19] `Eval<typename NODE, RndState RND>` is used to compile an efficient evaluation code for this expression. This trait is recursively applied on each node of the expression tree with the template parameter `NODE` corresponding to the type of the node and the enumerator `RND` describing the current rounding mode.

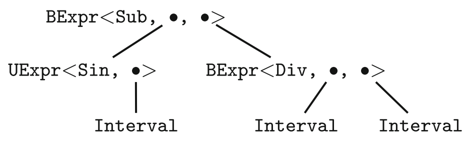


Fig. 4 Tree structure of the generated expression type of the interval expression $sin(X) - X/Y$

The first specialization of this trait class is the class `Eval<Interval, RndState RND>` which works on the leaf nodes (type `Interval`) of the tree whereat the rounding mode information `RND` is still variable. The implementation of

[6] Note that this pseudo code does not contain any exception handling for the reason of readability and the intermediate results for the upper bounds `s2` and `t2` are stored negated.

[7] In practice the class `Interval` should be implemented as a generic class with a template parameter for the type of the interval bounds.

this trait has a static function which returns the interval at this node and additionally store the current and unchanged rounding mode state (enumerator `RND`) in a static constant `rnd`.

The evaluation of the inner nodes is done by the trait specializations `Eval<BExpr<class OP,typename A, typename B>,RndState RND>` and `Eval<UExpr<class OP, typename A>, RndState RND>`, respectively. As well as the trait specialization for intervals these two traits have to store the information about the rounding mode which is active after the evaluation of this node. Normally this could be done by assigning the static constant `rndA` of the policy class `OP` to the static constant `rnd` of the trait. However if the rounding mode is not used by the policy class some additional work with template meta programming is necessary to compute the current rounding mode from the information of the child nodes.

The evaluation of an inner node using the trait specializations for binary expressions[8] `Eval<BExpr<class OP,typename A, typename B>,RndState RND>` performs as follows:

1. Use the trait `Eval<A, RND>` to mix in the evaluation of the first child node of type `A` using the current rounding mode `RND`.
2. Use the trait `Eval<B, Eval<A, RND>::rnd>` to mix in the evaluation of the second child node of type `B` using the rounding mode which was left by the trait for the evaluation of the first child node.
3. Mix in the necessary run time code for switching to the required rounding mode `OP::rndB`. This could be done by using a trait class `RndSet<RndState OLD, RndState NEW>` together with the information about the active rounding mode from the traits for the evaluation of the child nodes together with the new mode `OP::rndB`.
4. Mix in the function `OP::eval(a, b)` to evaluate the corresponding operation of the policy class using the results `a` and `b` from the evaluation of the child nodes.

The main concept of this approach is to entrain the available information about the active rounding mode during the transformation of the expression tree into the final run time code at compile time. Therefore, the trait class `RndSet<RndState OLD, RndState NEW>` could be used to mix in only the necessary run time code for switching the rounding mode. In cases like `OLD == NEW` or `NEW == UNUSED` no rounding mode switches are necessary and therefore a “dummy” trait with no functionality is used.

Just like in our first expression template implementation [20] the construction of the complex expression tree could be hidden from the user by operator overloading. Additionally, an overloaded assignment operator could perform the following steps to evaluate the expression with an optimized rounding strategy.

1. Store the primary rounding mode at run time.
2. Use the trait `Eval<T, UNKNOWN>` to mix in the evaluation of the expression tree `T`.
3. Restore the rounding mode at run time.

[8] The trait specialization for unary expressions is implemented in the same manner.

Note that the trait for the evaluation of the expression tree in Step 2 is parameterized with the enumerator `UNKNOWN` for the rounding mode state. This is necessary because this trait is mixed in at compile time and the stored rounding mode in Step 1 is only visible at run time. Hence, all the recursive calls of the trait class are uninformed about the rounding mode until they reach an operation of the expression which sets one of the four provided rounding modes. Afterwards, the corresponding rounding mode for the expression at run time is known at compile time.

3.2 Flexible tree structure

Our implementations in Sect. 3.1 as well as in [20] demonstrate, that expression templates are well suited to improve the performance of the computation of interval expressions without a disadvantage in the usability for the user. However, the developer of such a library is burdened with extra work to implement the additional classes and traits for all necessary kinds of operations (unary, binary, ternary, ...).

This extra work could be avoided by using a new feature of the upcoming C++ standard [3] called variadic templates [12]. Listing 2 demonstrates this new technique at a variadic tuple class which could be specialized at compile time with an arbitrary number of (different) types.

```
template<typename... ELEMENTS> class Tuple;

template<> class Tuple<> {
    /* zero-tuple implementation */
};

template<typename HEAD, typename... TAIL>
class Tuple : private Tuple<TAIL...> {
   HEAD head;
public:
    /* implementation */
};
```

Listing 2 A tuple class with variadic templates, see [12].

The ellipsis after the keyword `typename` of the template identifier `ELEMENTS` in the first line of Listing 2 labels this parameter as a template type parameter pack [12] which could be used with an arbitrary number of arguments. With the feasibility of “unpacking” a fixed number of arguments of the parameter pack it is possible to address each element using recursive template specializations. The last specialization of the tuple class in Listing 2 illustrates the separation of the first argument `HEAD` from the tail of the parameter pack to realize the tuple structure. Note that this specialization of the tuple class defines only one attribute of the type `HEAD`. The arguments of the tail are addressed by recursively inheriting from the tuple class containing these arguments until it reaches the zero-tuple implementation.

With this new concept it is now possible to replace all the different expression classes for unary, binary, ternary, ...operations by one universal class `Expr<class OP, typename... ARGS>` which is derived form the class `Tuple<ARGS...>` to hold an arbitrary number of arguments for the different exchangeable operations represented by the policy class `OP`. Furthermore the different specializations of the evaluation trait `Eval<typename NODE, RndState RND>` for the inner nodes are replaced by one specialization matching the new universal expression type `Expr<OP, ARGS...>` and performing the following steps:

1. Create an instance of type `Tuple<Interval, ..., Interval>` of the size[9] |`ARGS`|.
2. Evaluate each argument of the expression by recursively applying the evaluation trait on each single argument writing the result into the tuple of step 1.
3. Apply the tuple with all the results of the evaluated arguments to the policy `OP`.

Note that in Step 2 and 3 a recursive trait class `ElementAt<int N, class T>` is used to access the reference of the `N`th element of the tuple `T` at compile time by using the technique of "unpacking" template arguments [12]. Naturally this new trait specialization uses the same technique to compute the rounding mode at compile time as described in Sect. 3.1.

The new expression class together with only two specializations of the evaluation trait (one for intervals/leaf nodes, one for the new expression class/inner nodes) now offers the possibility to easily extend the expression template implementation with new operations and functions. This is done by providing a new policy class with the desired functionality together with an implemented function returning an instance of the new expression class distincted with this policy.

3.3 Automatic differentiation

The computation of the first or higher derivatives of a function is a common problem in scientific computing using interval arithmetic.

The most obvious approach to compute the derivative values is the symbolic differentiation which applies the well known rules of differentiation onto the expression to compute a formal expression of the derivative function. This is a feasible approach for problems with modest size but the limitation of symbolic processors is reached quickly [8].

Another method called numerical differentiation is the approximation of the derivative with difference quotients. In practice this approach suffers from quirks of the floating point numbers like truncation errors and roundoff errors.

Superior to these approaches is the automatic differentiation [21] which is a technique to compute the expression and the derivative together by applying the well known rules of differentiation. The difference to symbolic differentiation is that it propagates numerical values instead of formal expressions.

Commonly used implementations of automatic differentiation may be divided into two categories [5], the implementations like ADOL-C [13], FADBAD [4] or C-XSC [14] using operator overloading to compute the values of the function and

[9] The size of the template type parameter pack `ARGS` could be computed with the command `sizeof...(ARGS)` at compile time. The type `Tuple<Interval, ..., Interval>` then could be created by recursively "packing" [12] the template type parameter pack using template meta programming.

the derivative together as well as special tools like ADIC [6] or TAPENADE [15] applying the technique of source transformation to mix in the expressions to compute the derivatives. Both of these techniques are somehow inflexible in the manner that the operators are not overloaded for high order differentiation or additional tools are required for the source transformation.

We, however, use the expression tree to mix in the code for the differentiation using template meta programming techniques. The main concept is to apply the symbolic differentiation at compile time only onto the single operations of the expression tree to create the run time code for the automatic differentiation. This has two advantages, we can, in theory, compute derivatives of any order and secondly the differentiation of the operations is done with types. This means that the symbolic differentiation of an operation with a specified order is performed only once by the compiler. Afterwards for every occurrence of this operation with this specified order the distincted type is used.

To apply this approach onto our expression template implementation a restructuring of the expression tree and a modification of the evaluation traits is necessary. The leaf nodes of the tree are now represented by two different types `Interval` and `Var` standing for interval constants and interval variables. Additionally the evaluation traits matching the types `Interval` and `Var` are extended by a third template parameter[10] `N` of type `int` specifying the order of the differentiation and returning an instance of the type

$$\texttt{Tuple<}\underbrace{\texttt{Interval, ..., Interval}}_{\texttt{N}+1}\texttt{>}$$

filled with the appropriate values of the variable or constant together with the corresponding derivatives.

The more significant changes are related to the inner nodes of the tree. The policy classes which implement the functionality of the corresponding operation, see Listing 1, are replaced by new policy classes which only have a descriptive behavior. Listing 3 shows the definition of such a policy class for an interval multiplication. A `typedef` of nested template specifications of this new descriptive policy classes, applying the basic rules of differentiation, is used to define the derivative `df` of these operations.

```
template<class U, class V> struct Mul {
   // derivative: U * V' + U' * V
   typedef Add<Mul<U, typename V::df>, Mul<typename U::df, V>> df;
};
```

Listing 3 New policy class for an interval multiplication.

Additionally the descriptive policy class `Df<int I, int N>`, see Listing 4, is defined to handle the relation of the operation and its arguments. Thereby, the template parameters stands for the `N`th derivative of the `I`th argument.

[10] Besides the template parameters specifying the type of the node and the rounding mode.

```
template<int I, int N> struct Df {
   typedef Df<I,N+1> df;
};
```

Listing 4 Policy class describing the `N`th derivative of the `I`th argument.

With these descriptive policy classes it is easy to assemble the nested policy `OP` describing the operation required by an inner node. The following line defines the policy `OP` for a multiplication

$$\texttt{OP} := \texttt{Mul<Df<0,0>,Df<1,0>>}$$

as a multiplication of the values (`0`th derivative) of the first (index `0`) and second (index `1`) argument. Now the derivative of this operation

$$\texttt{OP'}=\texttt{OP::df} = \texttt{Add<Mul<Df<0,0>,Df<1,1>>,Mul<Df<0,1>,Df<1,0>>>}$$

can be easily deduced using the type `df` of the policy `OP` at compile time.

Up to now, the policy `OP` has only a descriptive behavior. The functionality of this policy is implemented by the static template function

```
template<class M> static Interval eval(M& m)
```

of the trait `EvalPolicy<class P>` specialized to match the descriptive policy classes like `Mul<class U, class V>` or `Df<int I, int N>`. Thereby the template parameter `M` stand for a matrix assembled by a tuple of tuples of intervals

```
Tuple<
      Tuple<Interval, …,Interval>,
      ...
      Tuple<Interval, …,Interval>
     >
```

containing the values and required derivatives of the subexpressions of the current operation or function, see Fig. 5. The specialization of the trait `EvalPolicy<class P>` for the type `Df<int I, int N>` then easily returns the reference of the interval at the `I`th row and the `N`th column of the matrix `m`. The specializations matching the other descriptive policies like `Add<class U, class V>`, `Mul<class U, class V>` or `Sin<class U, class V>` then naturally perform the corresponding operation or function on the return values of the recursive calls of the trait for the template parameters of this type.

The evaluation of an inner node `Expr<OP, ARGS…>` distincted with such a descriptive policy class `OP` is once again executed by the evaluation trait extended by a third template parameter `N` of type `int` specifying the order of the differentiation performing the following steps, see Figure 5:

1. Mix in the run time code to create an instance `m` of a matrix type of the size $|\texttt{ARGS}| \times (\texttt{N} + 1)$ assembled by a tuple of tuples of intervals.

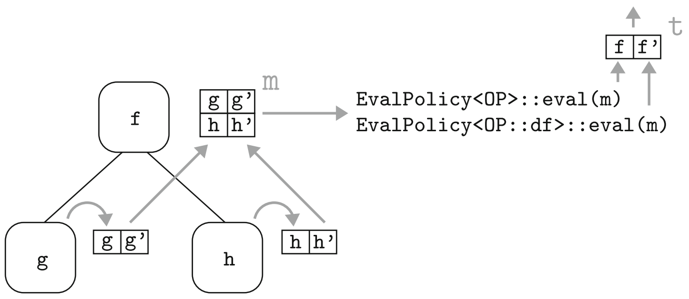


Fig. 5 Performed steps of the evaluation trait for an inner node computing the function `f(g(...),h(...))` together with its first derivative

2. Use the trait recursively to mix in the evaluation of the child nodes writing their results (value and derivatives) into the matrix `m`.
3. Mix in the run time code to create an instance `t` of a tuple of intervals of the size $\mathtt{N} + 1$.
4. Mix in the run time code for the evaluation of the operation and its derivatives by applying the matrix `m` onto the corresponding trait
   - `EvalPolicy<OP>::eval(m)`
   - `EvalPolicy<OP::df>::eval(m)`
   - `EvalPolicy<OP::df::df>::eval(m)`
   - ...

   writing their results into the tuple `t`.
5. Mix in the run time code to return the tuple `t`.

The use of operator overloading as well as a new type `Derivative<int N>` could be used to provide a domain specific language for interval arithmetic supporting automatic differentiation, see Listing 5 as an example.

```
typedef Derivative<2> DF2;

auto expr = sin(Var()) - Var() / Interval(100.0, 100.0);

DF2 res = expr(Interval(0.5, 1.5));
```

Listing 5 Computation of interval expression $sin(X) - X/[100, 100]$ for $X = [0.5, 1.5]$ using automatic differentiation up to the second derivative.

The class `Derivative<int N>` represents a tuple of intervals which stores the values of the expression up to the `N`th derivative. The evaluation of the constructed expression tree `expr`[11] is then performed by the overloaded function call operator and assignment operator to specify the value of the variable of the expression and to construct the required evaluation code, respectively.

[11] The C++0x keyword `auto` is a placeholder for the type of a variable which could be deduced by a compiler [16].

Note that in this exemplary implementation of the automatic differentiation the variable of an expression is represented by a single type `Var`. This means that the differentiation only works with one symbolic variable. But the extension of this approach for a partial differentiation with independent variables could easily be done by the introduction of a new generic type such as `Var<char c>` to use different variables like `Var<'x'>` or `Var<'y'>` [11]. And once again the technique to compute the required rounding mode described in Sect. 3.1 is also used in this approach.

4 Run time and compile time measurements

Due to the fact that several of the used C++ keywords and features belong to the upcoming C++ standard, the compiler support for this work is not very good at the moment. To our knowledge, the GNU C++ Compiler in version 4.4 or above is the only compiler which can handle all of the used C++0x keywords and features. Hence, we have used the GNU compiler in version 4.4.3 with option `-O2` on a Linux system with the Intel Core 2 Quad Processor Q9550 (12M Cache, 2.83 GHz, 1333 MHz FSB) for the measurements.

Figure 6 shows a comparison of the average run time of our new expression template approach against our prior implementation [20] for 100,000 repetitions of interval expressions with different length. Note that the run time of our new approach is measured for the computation of the function value as well as for the computation of the first and second derivative whereas the run time of our prior approach is only measured for the computation of the function value. As we can see, the new approach results in a further performance increase for longer expressions.

To test the approach in a realistic environment we used our implementation in a interval Newton method for the function

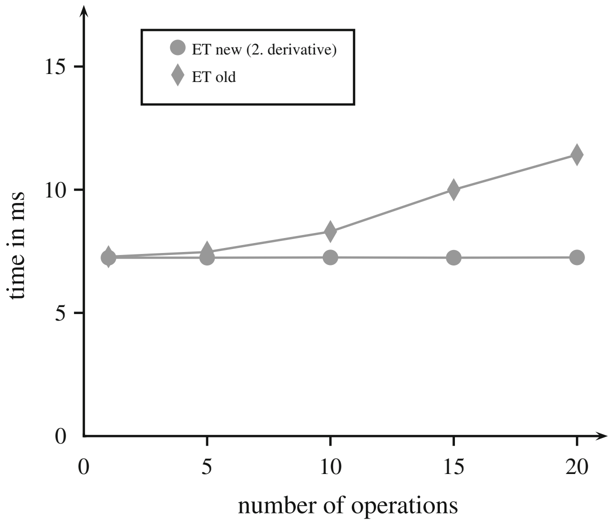


Fig. 6 Performance comparison of our new expression template approach against our prior implementation [20] for interval expressions with different length

$$x^{14} - 539.25 * x^{12} + 60033.8 * x^{10} - 1.77574e6 * x^8$$
$$+1.70316e7 * x^6 - 5.50378e7 * x^4 + 4.87225e7 * x^2 - 9e6$$

and measured a speedup of about 20% compared to a similar implementation in filib++.

On the other hand we noticed that the compile time suffers not much. For the case study with the interval Newton method we have measured a increase of the compile time of about 5.6% compared to the implementation using filib++.

5 Related work

In [11] Gil and Gutterman used expression templates and template meta programming to perform symbolic differentiation at compile time whereas Aubert, Césaré and Pironneau used expression templates to reduce the number of loops, temporaries and copies while computing the first order partial derivatives of an expression [2].

6 Conclusion and further research

Our first attempt [20] of using expression templates for an efficient interval arithmetic library achieved promising results and points us to a new perception of how to implement interval arithmetic in C++. In this paper we enhanced this approach in providing a flexible tree structure which is easy to extend with new operations and functions. Furthermore we used template meta programming to identify only the necessary rounding mode switches during the evaluation of an interval expression which led to an additional performance increase compared to [20]. Additionally the availability of an explicitly visible expression tree during compile time offered us the possibility to mix in the required run time code to perform automatic differentiation of arbitrary order.

Supplementary we showed how an expression template implementation as well as their usage could benefit form new language features of the upcoming C++ standard [3].

Further investigations are planned to improve interval arithmetic and scientific computing using expression templates and template meta programming especially in the connection to new features of the coming C++ standard.

References

1. Alexandrescu A (2001) Modern C++ design: generic programming and design patterns applied. Addison-Wesley Longman Publishing Co. Inc., Boston
2. Aubert P, Di Césaré N, Pironneau O (2001) Automatic differentiation in C++ using expression templates and application to a flow control problem. Comput Vis. Sci. 3:197–208
3. Becker P (2011) Working Draft, Standard for Programming Language C++. Tech. Rep. N3242= 11-0012, ISO/IEC JTC1/SC22/WG21
4. Bendtsen C, Stauning O (1996) FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark

5. Bischof CH, Bücker HM (2000) Computing derivatives of computer programs. In: Grotendorst J (ed) Modern methods and algorithms of quantum chemistry: proceedings, 2nd edn. NIC Series, vol 3, pp 315–327. NIC-Directors, Jülich
6. Bischof CH, Roh L, Mauer A (1997) ADIC—An extensible automatic differentiation tool for ANSI-C. Softw Pract Exp 27(12):1427–1456
7. Boost Interval Arithmetic Library (2011). http://www.boost.org/doc/libs/1_46_1/libs/numeric/interval/doc/interval.htm
8. Corliss GF, Griewank A (1993) Operator Overloading as an Enabling Technology for Automatic Differentiation. Tech. Rep. CRPC-TR93431, Center for Research on Parallel Computation, Rice University, Houston
9. CRlibm-Correctly Rounded mathematical library (2011). http://lipforge.ens-lyon.fr/www/crlibm/
10. Daramy-Loirat C, Defour D, de Dinechin F, Gallet M, Gast N, Quirin Lauter C, Muller JM (2009) CR-LIBM A library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/frs/download.php/153/crlibm-1.0beta3.pdf
11. Gil J, Gutterman Z (1998) Compile time symbolic derivation with C++ templates. In: Proceedings of the 4th conference on USENIX conference on object-oriented technologies and systems, vol 4, COOTS'98, pp 18–18. USENIX Association, Berkeley
12. Gregor D, Järvi J, Powell G (2006) Variadic templates (revision 3). Tech. Rep. N2080=06-0150, ISO/IEC JTC1/SC22/WG21
13. Griewank A, Juedes D, Utke J (1996) Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. ACM Trans Math Softw 22(2):131–167
14. Hammer R, Ratz D, Kulisch U, Hocks M (1997) C++ Toolbox for Verified Computing I: Basic Numerical Problems. Springer-Verlag New York Inc., Secaucus
15. Hascoët L, Pascual V (2004) TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis
16. Järvi J, Stroustrup B, Reis GD (2004) Decltype and auto (revision 4). Tech. Rep. N1705=04-0145, ISO/IEC JTC1/SC22/WG21
17. Lambov B (2008) Interval arithmetic using SSE-2. Lecture Notes in Computer Science, vol 5045, pp 102–113
18. Lerch M, Tischler G, Wolff von Gudenberg J, Hofschuster W, Krämer W (2006) Filib++, a fast interval library supporting containment computations. ACM Trans Math Softw 32(2):299–324
19. Myers N (1995) A new and useful template technique: traits. C++ Report 7(5):32–35. Reprinted in Lippman SB (ed) 1996 C++ Gems. SIGS publications, Inc., New York, NY, USA
20. Nehmeier M, Wolff von Gudenberg J (2011) filib++, Expression Templates and the Coming Interval Standard. Reliab Comput 15(4):312–320
21. Rall LB (1981) Automatic differentiation: techniques and applications. Lecture Notes in Computer Science, vol 120. Springer, Berlin
22. Veldhuizen T (1995) Expression templates. C++ Report 7(5):26–31. Reprinted in Lippman SB (ed) 1996 C++ Gems. SIGS publications, Inc., New York, NY, USA
23. Veldhuizen T (1995) Using C++ template metaprograms. C++ Report 7(4):36–43. Reprinted in Lippman SB (ed) 1996 C++ Gems. SIGS publications, Inc., New York, NY, USA