Automatic Differentiation in C++ using Expression Templates and Application to a Flow Control Problem

Pierre Aubert\textsuperscript{1}, Nicolas Di Césare\textsuperscript{2}, Olivier Pironneau\textsuperscript{2}

\textsuperscript{1} UMR 5585, Centre de mathématiques, I.N.S.A., Lyon, France.
\textsuperscript{2} Laboratoire d'Analyse Numérique, Université Pierre et Marie CURIE, Paris VI, France.

June 7, 1999

Abstract. This work deals with an implementation of automatic differentiation of C++ computer programs in forward mode using operator overloading and expression templates. We apply this tool to a flow control problem: minimize the drag of a cylinder, in subsonic unsteady turbulent flow, by controlling the boundary condition of the cylinder. We report on the efficiency of such implementation and its obvious advantage: the ability to perform sensitivity analysis without touching the source of the computer program by simply adding a library to it.

1 Introduction

Derivatives of functions can be computed exactly not only by hand but also by computers. Commercial software packages such as MAPLE [19], MATHEMATICA [20] have derivation operators implemented by formal calculus techniques. In [3] Griewank implemented a new idea, now called automatic differentiation (AD) of computational approximations to functions [6] and used first in his package ADOL-C [7] whereby formal calculus is used on each instruction of the computer program which describes the function. Thus if a function is described by its computer implementation, it can be differentiated exactly and automatically.

Applications of AD are numerous but it is control theory which has benefited most from it. Indeed gradient methods for control problems require a sensitivity analysis of the cost function $J(v)$ with respect to the control variable $v$, i.e., the computation of $J'_v$. In many cases the formal description of $J$ is so complex that an analytic computation of $J'$ is a formidable task. The control of flows is one such case.

In [12], B. Mohammadi used the AD software, ODYSSEE [14], to control unsteadiness in a flow around a cylinder modeled by the Navier-Stokes equations with a $k$–$\varepsilon$ turbulence model; the control is the flux at each time step of two small jets in the boundary of the cylinder. In principle the process is automatic: the Fortran program implementing the computation of the drag of the cylinder and the flow solver is an input to ODYSSEE which gives on output the Fortran program which computes the derivative of the drag with respect to the control. In practice one has to monitor the process to avoid out of memory faults.

On the other hand ODYSSEE gives a proper implementation of the derivatives because it uses the so-called reverse mode, which corresponds in control theory to the use of adjoint state variables.

Operator overloading has made AD very easy to implement. For instance every time the multiplication of $x$ by $y$ is needed, the chain rule $x \times dy + dx \times y$ is performed in the background by overloading the operator $\times$. In C++ this can be done in a class of differentiable floats, called say $\text{Fad}$ (Forward automatic differentiation), so that by simply changing in the source of the flow solver the C++ reserved word $\texttt{float}$ (or double) by $\text{Fad<float>}$ (or $\text{Fad<double>}$) one generates a C++ program which also computes the derivative of any variable with respect to any parameter in the program.

This however corresponds to the forward (or direct) mode and not to the reverse (or adjoint) mode. Therefore it will not be efficient if the number of control parameters is large, say larger than twenty. But for two dimensional flow control problems the control variables are usually not so numerous and then the method can be used with a good chance of efficiency. Efficiency is indeed the issue and the purpose of this report, because operator overloading multiplies the number of indirect addressings and there can be an explosion of the number of temporary
variables generated by the compiler. But thanks to T. Veldhuisen and his expression templates [17], efficient implementations are now possible.

The idea of using operator overloading for AD can be traced in [1], [7] and [2]. It has been used extensively in [8] for the computation of Taylor series of computer functions automatically and we wish to acknowledge the fact it is this later work which has instigated this study. It is also by discussing with B. Mohammadi and pondering over the know how that is now needed to obtain efficient AD Fortran programs with the reverse mode strategy that we thought of going back one step and review the forward mode approach.

Thus our goal is to reproduce Mohammadi’s flow control study [12] with the forward mode approach and to report of the efficiency of implementations by expression templates.

The paper is organized as follows: in Section 2, we state the problem and its discretization by a finite volume/finite element method with an explicit four steps Runge-Kutta (RK4) time algorithm. In section 3 we recall the concepts of AD and present our implementation in section 4. Finally we present the numerical results in section 5.

The conclusion is that the control problem has been solved in a reasonably efficient manner with this implementation of AD and Mohammadi’s results have been reproduced. The quality of the compiler seems crucial and so benchmarking it prior to implementation is a must.

2 Flow Control Problem

2.1 The Compressible Navier-Stokes Equations

Consider a compressible fluid with the following state variables: \( \rho \) the density, \( \mathbf{u} = (u_1, u_2) \) the velocity, \( p \) the pressure, \( E = T + \frac{\| \mathbf{u} \|^2}{2} \) the total energy per unit of volume, where \( T \) is the temperature, \( \kappa \) the turbulent kinetic energy and \( \varepsilon \) rate of dissipated turbulent energy. Let \( W = (\rho, \rho u_1, \rho u_2, \rho E, p, \rho \kappa, \rho \varepsilon)^T \) be the state variables and \( F(W) = (F_i(W), F_2(W))^T \) be the flux (advective operator) defined by:

\[
F_i(W) = \begin{pmatrix}
\rho u_i \\
\rho u_i u_1 + \delta_{i1} p \\
\rho u_i u_2 + \delta_{i2} p \\
u_i(E + p) \\
\rho u_i k \\
\rho u_i \varepsilon
\end{pmatrix}
\]

with viscous terms equal to

\[
N(W, \nabla W) = \begin{pmatrix}
O \\
(\mu + \mu_t) \nabla \mathbf{u} + (\kappa + \kappa_t) \nabla \mathbf{E} - \frac{\| \mathbf{u} \|^2}{2} \\
(\mu + \mu_v) \nabla k \\
(\mu + \mu_c \mu_t) \nabla \varepsilon
\end{pmatrix}
\]

and \( S \) the source terms equals to:

\[
S(W) = (0, f_1, f_2, f_3, S_k, S_\varepsilon)^T.
\]

With smooth initial conditions \( \rho_0, \mathbf{u}_0 \) and \( p_0 \) and smooth boundary conditions, \( \mathbf{u} \) and \( p \) given on boundaries and \( \rho \) given on the inflow boundaries only \( (\mathbf{u} \cdot \mathbf{n} < 0) \), the problem is well posed.

2.2 Discretizations

We use an explicit RK4 time integration scheme. The equations are discretized by a finite volume Galerkin upwind technique using a Roe Riemann solver for the convective part of the equation and a standard Galerkin method for the viscous terms (Dervieux, Desideri [5] and Mohammadi [9]).

2.3 Boundary Conditions

Natural boundary conditions could be homogeneous Dirichlet conditions for \( k \) and non-homogeneous Dirichlet conditions for \( \varepsilon \). However, the \( k - \varepsilon \) model is not valid near the solid boundaries because the local Reynolds number is not large, so a low Reynolds modification of the coefficient of the model would be necessary. Instead, we use wall laws for the solid boundary. A justification of this approach can be found in [18] or [11].

The wall law approach involves homogeneous non linear Frechet conditions, for the velocity and the temperature near solid walls.

The inflow and outflow boundaries are treated by upwinding techniques:

\[
\int_{\Gamma_{in/out}} F(W)^T \mathbf{n} \, d\sigma =
\]

\[
\int_{\Gamma_{in/out}} (A^T W + A W_{in/out})^T \mathbf{n} \, d\sigma
\]

where \( W_{in/out} \) is the external value given by the flow configuration.

2.4 Control law

The flow control behind a cylinder, at Reynolds 42000 and inflow Mach number of 0.4, is a difficult test
case. B. Mohammadi and G. Medic showed in [10] that their wall laws and \( k - \varepsilon \) equations are feasible for such a flow.

There are three ideas to control the unsteadiness of the flow. The first one is to give an angular velocity to the cylinder (figure 1), the second one is to put rotating modules behind the cylinder (figure 2) and the third one is to use injection/suction modules (jets) on the boundary of the cylinder (figure 3).

In this study we considered the last one only. We have put the controlled jets behind the cylinder for \( 30^\circ \leq \theta \leq 40^\circ \). The boundary condition is defined as an inflow/outflow conditions

\[
\int_{\Gamma_{cur}} F(W)^T \mathbf{n} \, d\sigma = \int_{\Gamma_{cur}} (A^+ W + A^- W_{ctrl}(\alpha, \beta))^T \mathbf{n} \, d\sigma,
\]

where \( \Gamma_{cur} \) is the part of the cylinder boundary where the control is applied and \( \alpha, \beta \) are the scalar controls. In contrast with the inflow boundary conditions, we control the sign and the norm of the velocity field \( u_{ctrl} \), by means of \( \alpha \) and \( \beta \).

2.5 Optimization

The jets are controlled to minimize the drag coefficient. Therefore, the optimization problem reduces to the minimization of the drag function \( C_D \) with respect to the control coefficients \( \alpha, \beta \)

\[
V(\alpha, \beta) = \min_{\alpha,\beta} J(\alpha, \beta)
= \min_{\alpha,\beta} \int_0^T C_D(\alpha(t), \beta(t))
\]

where

\[
C_D(\alpha, \beta) = \frac{1}{2\rho_{\infty} \| u_{\infty} \|^2} \int_{\Gamma_{cyl}} u_{\infty} \left( p I - (\mu + \mu_t) S \right) \mathbf{n} \, d\sigma,
\]

\( u_{\infty} \) and \( \rho_{\infty} \) are the velocity vector and the density at the inflow boundary, \( \Gamma_{cyl} \) is the cylinder boundary.

2.6 Optimization Algorithm

One time iteration in the flow computation is defined by the following problem, corresponding to the discretization of the flow equations (1)

\[
W_m(\alpha, \beta) = W_{m-1}(\alpha, \beta)
+ \Delta t G_{m-1}(W_{m-1}(\alpha, \beta)).
\]

We use a gradient algorithm for the optimization loop to minimize the drag at each step.

3 Derivative Computation

There are at least three ways to compute derivatives in \( \alpha \) and \( \beta \). The first one is analytic, i.e. by hand or by using one of the symbolic differentiation packages [19,20]. It is powerful but perhaps laborious also. The second one is numerical, i.e. by finite difference approximations but it leads to truncation error and inaccurate results. The third way is by automatic differentiation of the computer program. It is quite efficient and simple especially if operator overloading is used as we shall demonstrate.

3.1 Automatic Differentiation

We consider a program with \( N \) variables \((x_i)_{1 \leq i \leq N}\) and we denote it by \( P(x_1, ..., x_N) \). This program has \( m \) output variables \((x_i)_{i \in D_{out}}\), where

\[
D_{out} \subset \{1, ..., N\}
\]

is the set of output variables.

We will differentiate the program \( P \) with respect of the \( n \) first variables \((x_i)_{1 \leq i \leq n}\). These variables are called independent variables.
A variable \( x_i, i > n \), becomes active when an independent or an active one is assigned to it.

If \( D \) is a subset of \( \{1, \ldots, N\} \), we denote by \( x_D = (x_i)_{i \in D} \). We assume that the program is made of \( K \) assignment instructions sequentially ordered. Let \((D_k)_{1 \leq k \leq K} \) be \( K \) subsets of \( \{1, \ldots, N\} \). Thus, our program \( P \) can be view as follow:

**Program 1** \( P(x_1, \ldots, x_N) \)

for \( k = 1 \) to \( K \) do 
\[
 x_{\mu_k} = \varphi_k (x_{\mu_k})
\]
end for

where \((\varphi_k)_{1 \leq k \leq K} \) are functions using variables \( x_{D_k} \). For example, we consider a simple case.

**Program 2** \( P(x_1, x_2, x_3, x_4, x_5) \)

\[
\begin{align*}
 x_3 &= x_1 x_2 & \mu_1 &= 3 & \varphi_1(x) &= x^2 & D_1 &= 2 \\
 x_4 &= \sin(x_3) & \mu_2 &= 4 & \varphi_2(x) &= \sin(x) & D_2 &= 3 \\
 x_5 &= 2x_1 x_3 & \mu_3 &= 5 & \varphi_3(x, y) &= 2xy & D_3 &= 1.3 \\
 x_6 &= x_5 / x_4 & \mu_4 &= 5 & \varphi_4(x, y) &= x / y & D_4 &= 5.4
\end{align*}
\]

This produces the following table:

<table>
<thead>
<tr>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( x_6 )</th>
<th>( \mu_k )</th>
<th>( \varphi_k )</th>
<th>( D_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 x_2 )</td>
<td>( \sin(x_3) )</td>
<td>( 2x_1 x_3 )</td>
<td>( \frac{x_5}{x_4} )</td>
<td>3</td>
<td>( x^2 )</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3.3 Implementation of the forward mode

There are two approaches to implement automatic differentiation:

1. A pre-treatment of the source code that provides a derived code. ODYSSEE [14], Adifor [3] and Adic [4] are well known software using this approach. This is easier for language with simple data structures like Fortran77 than for languages with complex data structures like C or C++.

2. Operator overloading features(C++). ADOL-C [7], and FADBAD [2] are using this approach.

Several persons have already used the overloading technique. The first ones were A. Griewank et al. with ADOL-C [7], later C. Bendtsen and O. Stauning with FADBAD [2].

M. Masmoudi and M. Grundman presented this technique in July 97. We chose it for it simplicity and also because the evolution of C++ compilers were about to provide us with more efficient numerical tools (expression templates [17] and std::valarray<>... ) in a near future. And actually in march-april 98, when a free compiler EGCS1 allowed us to compile expression templates, we decided to use this technique in our classes. However, the implementation was not as simple. But the performance increase justify the extra work, as shown in figure 5.

ADOL-C and FADBAD also implement the reverse mode that is used for problems with a large number of independent variables.

4 Implementation in C++

4.1 Introduction

Let

\[
\mathcal{F}_{arith} = \{ + , - , \ast , / , +(unary), \} - (unary), ++, -- \}
\]

be the set of arithmetic operators,

\[
\mathcal{F}_{logical} = \{ !, ==, <, >), <, =, \}
\]

be the set of logical operators and

\[
\mathcal{F}_{math} = \{ sqrt, log, pow, \}
\]

be the set of mathematical functions defined in C++.

This implementation of the forward mode is based on the operator overloading feature of the C++ [15] which allows us to rewrite the basic operations of sets defined in (5,6) and functions of set defined in (7) available for the classical real types (double, float) for our Fad class.

1 since version 1.0.3, http://egcs.cygnus.com/
Listing 3 Forward Automatic Differentiation Class

```cpp
template <class T> class Fad {
    protected:
        T val;
        Vector<T> dx_;  
    public:
        ...
};
```

As shown in listing 3, this class has two data: the first one is the value `val` that is always set to an initial value, either zero given by the default constructor, `Fad()`, or a value given by the other constructors: `Fad(const T & value)` or `Fad(const Fad<T> & fad)`. The second one is the vector `dx_` of partial derivatives. The size of this vector, given by the member function `size()`, is different from zero when the variable is an active or an independent one.

A variable becomes independent when we call the member function `diff(const int n)` that set the number of `independent` variables to `n` and set the gradient `dx_` to the `-`th element of the canonical basis.

The member functions `val()` and `dx(int i)` are used to access respectively to the value and the `-`th partial derivatives.

Before the presentation of the implementation with expression templates, we have to introduce a technique, called traits, to solve type promotion problems.

4.2 traits and automatic return type determination

Traits were introduced by N. Myers [13] to solve stream problems. The most interesting use of traits for scientific computing is type promotion for templates classes. We consider the addition of two Fad of different types:

```cpp
Fad<TYPE> = Fad<double>
    + Fad<stl::complex<float>
```

The problem is to automatically know the return type `TYPE`. For that, we use C promotion rules and mathematical promotion rules:

- C rules:
  ```cpp
  float + double →double,
  int + float →float,...
  ```
- Mathematical rules:
  ```cpp
  double + complex →complex,...
  ```
- Mixed rules:
  ```cpp
  double+complex<float> →complex<double>,...
  ```

and we apply the rules to `Fad<>` calculation.

Thus we implement these rules using the template specialization feature. The field `promote` is used to provide the type of the result of any binary operators on the types `L` and `R`. For the general case, we cannot specify the return type which could be different than `L` or `R`, thus we introduce an empty struct `returnTypeNotDefined` as the return type. When

```cpp
Listing 4 NumericalTraits, General Definition

```cpp
struct returnTypeNotDefined {};
```
```cpp
template <class L, class R>
struct NumericalTraits {
    typedef returnTypeNotDefined promote;
};
```

the compiler do not find a matching partial specialization, it outputs `returnTypeNotDefined` which should be clear enough. Specializations are done in listing 5.

```cpp
Listing 5 Specialization

template <> struct NumericalTraits<stl::complex<float>,double> {
    typedef stl::complex<double> promote;
};
```

Indeed, an operation of `double and complex<float>` will give a `complex<double>`. We define in annex A an automatic way to define specializations, using more meta-programming [16] and intermediate templates.

As shown in listing 6, the type `value_type` for a binary operators is determined using the numerical traits `NumericalTraits<>`.

```cpp
Listing 6 Type Promotion for Binary Operators

template <class L, class R>
class BinaryOperator {
    public:
        typedef typename L::value_type value_type_L;
        typedef typename R::value_type value_type_R;
        typedef typename NumericalTraits<value_type_L,
                                        value_type_R>::promote value_type;
    ...
};
```
The keyword `typename` means that the type fields `value_type` of `L` and `R` are types.

Now with the `NumericalTraits`, we can explore the power of expression templates techniques.

### 4.3 expression templates techniques [17]

The expression templates techniques were introduced by T. Veldhuizen in 1995 for vectors and array computations. Using these techniques, he provided an active library called Blitz++\(^2\) that equals Fortran performances on several Unix platforms (IBM, HP, SGI, LINUX workstations and Cray T3E). The vector structure of our classes, incited us to use these techniques.

In listing 7, we write the simplest way to implement the addition operator.

**Listing 7 Classical addition operator**

```cpp
template <class T> inline Fad<T>
operator+(const Fad<T> &x1, const Fad<T> &x2)
{ // a temporary is created here
    Fad<T> tmp(x1.size());

    for(int i=0; i<tmp.size(); ++i) // Loop
        tmp.dx(i) = x1.dx(i) + x2.dx(i);
    tmp.val() = x1.val() + x2.val();

    return tmp;
}
```

We consider the operations \(A = B + C + D\), where \(A, B, C\) and \(D\) are `Fad<>` variables. The previous implementation introduces temporaries for \(B + C\) and \(B + C + D\) and also copies the results.

The basic idea of expression templates is to build a binary tree composed of elements of \(\mathcal{F}_{\text{arith}} \cup \mathcal{F}_{\text{logical}} \cup \mathcal{F}_{\text{math}}\) to simulate a n-ary operator (if the tree has \(n\) nodes). This tree is evaluated when the assignment operator is called. By example the following operations \(A + B \times \log(C)\) produce the tree of figure 4:

```
        (+)
         / \  
        A  (*)
         / \  
        B (log)  C
```

**Fig. 4. expression templates tree.**

\(^2\) http://monet.uwaterloo.ca/blitz

To build the tree, we define auxiliary classes for all operators in \(\mathcal{F}_{\text{arith}} \cup \mathcal{F}_{\text{logical}}\) and all functions in \(\mathcal{F}_{\text{math}}\). In these classes, we overload the member functions `val()` and `dx(int i)` corresponding to the operator or to the function.

For the addition, we define the `FadBinaryAdd` class, in listing 8, where `value_type` is defined as described in listing 6.

**Listing 8 Addition Class**

```cpp
template <class L, class R>
class FadBinaryAdd
{
    protected:
        const L& left_; const R& right_;
    public:
        // define value_type using NumericalTraits
        FadBinaryAdd(const L& left, const R& right) :
            left_(left), right_(right) {};

        const value_type val() const
            {return left_.val() + right_.val();}
        const value_type dx(int i) const
            {return left_.dx(i) + right_.dx(i);}
};
```

That produces the addition operator of listing 9.

**Listing 9 expression templates for addition operator**

```cpp
template <class L, class R>
inline FadBinaryAdd<L,R>
operator+( const L& left, const R & right) {
    return FadBinaryAdd<L,R>(left,right);}
```

The template arguments `L` and `R` are either a class operator like

(FadBinary [Add, Minus, Mul, Div] `<,>`) or a class function

(FadFunc [Sqrt, Log, Pow, ...] `<<`) that can be viewed as a node of the tree, or a `Fad<>` class, that can be viewed as a leaf.

One important thing to understand is that these classes, and their associated operators, do not compute anything. It is just a automatic way to build a tree that will be evaluated by the assignment operator.

This template feature allows us to use the compiler to instantiate all the needed classes. For example the result of `B + C + D` will be of type:
This approach addressed the three problems of the previous class that lead to a loss of performances:
- the number of temporaries is minimized,
- the number of copies is minimized,
- the number of loops is minimized.

Thus we have to add a new assignment operator, in the class `Fad<T>`, taking as argument a variable of type `FadBinaryAdd<L, R>` (see listing 10). It should be noted that a copy constructor is not needed.

Listing 10 FadBinaryAdd Assignment Operator

```cpp
template <class L, class R> Fad<T> &
Fad<T>::operator=(const FadBinaryAdd<L, R>& rhs)
{
    for(int i=0; i<rhs.size(); ++i)
        dx_[i] = rhs.dx(i);
    val_ = rhs.val();
    return *this;
}
```

In this way, we have also to define assignment operators for each element of \( \mathcal{F}_{\text{arith}} \cup \mathcal{F}_{\text{logical}} \cup \mathcal{F}_{\text{math}} \), i.e. for the following classes, `FadBinary[Minus, Mult, Div]<,>` and `FadFunc[Sqrt, Log, Pow, ...]`. It would be tedious and we can remark that all these classes define the member functions `val()` and `dx()`. Usually when objects have common characteristics, as these member functions, people use an abstract class to provide a common interface to the objects with respect to these characteristics. And we define only one assignment operator with this abstract class. Then the auxiliary classes have to be derived from this abstract class in order to be assign to `Fad<T>`. Unfortunately, the use of abstract class implies time penalty to access to the virtual members, because the internal call mechanism, for these members, uses the well known virtual table to recognize which derived class is used. In our case, the virtual members do not compute anything that could make up for the use of a virtual function.

The solution of this problem is the definition of a template interface `FadExpr<T>` (see listing 11), in real words an expression of `Fad<T>`. It will be the common return type of all operations, where `T` is an operation class like `FadBinaryMinus` or `FadFuncSqrt`. As shown in listing 12, we only change the return type when we define the operator. Finally, in listing 13, we only define one assignment operator with `FadExpr<T>` class.

This kind of operator is called member template operator in the language, and that is not supported by old compilers. This technique implies the addition of a template level at each operation and it increases the compilation time.

We also need to define an auxiliary class `FadCst<>`, that has also a template interface, for operations with the basic types of the language (float, double, ... ) that can be view as constants. In listing 14, we define the `FadCst` class, where the constructor `FadCst(const T & value)` initialize constant `value` to a given value. The member function `val()` returns the value of the constant and the member function `dx(int i)` returns zero for all `i`.

Listing 11 Fad Expression

```cpp
template <class T> class FadExpr {
protected:
    const T fadexpr_; // can't be a reference
public:
    typedef typename T::value_type value_type;

    explicit FadExpr(const T& fadexpr)
    : fadexpr_(fadexpr) {}

    const value_type val() const
    { return fadexpr_.val(); }
    const value_type dx(int i) const
    { return fadexpr_.dx(i); }

    int size() const { return fadexpr_.size(); }
};
```

Listing 12 FadBinaryAdd Expression templates addition operator

```cpp
template<class E1, class E2>
inline FadExpr< FadBinaryAdd< FadExpr<E1>,
                   FadExpr<E2> > >
    operator+ (const FadExpr<E1> &v,
                const FadExpr<E2> &w) {
        typedef FadBinaryAdd< FadExpr<E1>,
                             FadExpr<E2> > expr_t;
        return FadExpr<expr_t>(expr_t (v, w));
    }
```

Listing 13 Fad Expression Assignment Operator

```cpp
template <class T> template <class ExprT>
inline Fad<T> &
Fad<T>::operator=(const FadExpr<ExprT>& expr)
{
    for(int i=0; i<dx.size(); ++i)
        dx_[i] = expr.dx(i);
    val_ = expr.val();

    return *this;
}
```
Listing 14 Fad Constant Class

```cpp
template < class T > class FadCst {
  protected:
    const T constant_;
  public:
    ...;
};
```

At the end, the nodes of the tree are only of type FadExpr and the leaves are either of type Fad, either of type FadCst.

Finally, we define operators and functions specializations for all combinations of types i.e.

FadExpr<> OP FadExpr<> , Fad<> OP FadCst<> , FadCst<> OP Fad<> , FadExpr<> OP FadCst<> , FadCst<> OP FadExpr<> , FadExpr<> OP Fad<> , Fad<> OP Fad<> , Fad<> OP FadExpr<>

where OP is a binary operator or a binary function. Some examples are shown in the first annex.

5 Results

5.1 Benchmark

In the first part of this section, we show how the use of expression templates is powerful compared to classical overloading. In the second one, we compare our class to Adol-c 1.8 and FadBad 2.0 that implement another technique using overloading.

5.1.1 expression templates performances

We have done four computations (figure 5) to compare differentiation “by hand” (H) and automatic differentiation (AD):

- “By hand”
  - (H1): using something that would have been provided by a derivative code generator. It is the reference computation time. The three other computation times are then divided by this one.
  - (H2): using the member functions of the class named Fad<> , to test inlining capacity of the compilers.

- by automatic differentiation
  - (AD1): using simple overloading method. This should test the capacity of compiler to eliminate temporary.
  - (AD2): using expression templates.

All these computations are done by increasing the number of independent variables from 1 to 20. In these tests we compute the expression,

\[ 2.0\times(x_1\times x_2) + 3 + x_3/ x_5 + 4.7 \]

and it derivatives with respect to the independent variables.

The performances are quite dependent on the compiler. The figures 5, 7, 6 and 8 show the results for two well known compilers, EGCS and KCC. EGCS is an extension of the GNU C++ compiler, which is in the public domain. KCC, the KAI C++3 compiler, is commercial.

These compilers are the first ones available for most Unix platforms (June 98), supporting expression templates and member template features.

![Graph 5](image-url)  
**Fig. 5.** Computing with EGCS 1.1.2 : ratio between the computation times of (H2), (AD1), (AD2) and the reference time (H1).

![Graph 6](image-url)  
**Fig. 6.** Same as figure 5 with KCC 3.3g.

The use of expression templates decreases substantially the computation time, by a factor of 3 or 4. It is still not enough, compared to the computation time with the member functions, the ratio is still

3 version 3.3g. [http://www.kai.com/](http://www.kai.com/)
greater than 2. Thus, we have to decrease this ratio and compilers have to decrease the ratio between the call of a function and the access to a pointer.

On the other side, it increases the compilation time and also the memory required to compile. To give an idea, the compilation of the code described in the next section 5.2 takes one hour and requires 200 Mo. A lot of work has to be done to decrease these two problems. At the level of the expression templates technique, T. Veldhuizen is working on new techniques that decrease by a factor of 2 or 3 the compilation time and also the memory requirements. The compilers have to be optimized in the analysis part of expression templates, more precisely in what is called in EGCS “global common subexpressions elimination” (G.C.S.E.\textsuperscript{4}).

\textsuperscript{4} http://egcs.cygnus.com/gcse.html

5.1.2 Comparison with ADOL-C [7] and FadBAD [2]

In order to provide a more understandable test, we have also done computations with the forward mode of ADOL-C 1.8 and FadBAD 2.0. These computations have been done only with EGCS because ADOL-C required several changes to be compilable with the KCC.

ADOL-C also tries to minimize the number of temporaries and loops introduced by the classical overloading technique. But it is managed using of pointers and not auxiliary template classes. FadBAD uses the classical overloading approach.

\textbf{Listing 15} ADOL-C, FadBAD and expression templates

\begin{verbatim}
// ADOL-C
trace_on(1);
y = 0.;
for(i=0; i<n; i++) {
    tmp <<= xp[i];
    y = ((y*tmp) + (tmp/tmp)) - ((tmp*tmp) + (tmp - tmp));
}
y >>= yp;
trace_off();
forward(1,1,n,0,xp,g);// gradient evaluation

// Fad\double gradients and values computed
// at the same time
y = 0.;
for(i=0; i<n; i++) {
    Fad\double tmp(xp[i]);
    tmp.diff(i,n);
    y = ((y*tmp) + (tmp/tmp)) - ((tmp*tmp) + (tmp - tmp));
}

// F\double gradients and values computed
// at the same time
y = 0.;
for(i=0; i<n; i++) {
    F\double tmp(xp[i]);
    tmp.diff(i,n);
    y = ((y*tmp) + (tmp/tmp)) - ((tmp*tmp) + (tmp - tmp));
}
\end{verbatim}

In the test, we define a vector of independent variables and we perform several arithmetic operations on this vector that are accumulated in a single variable y. We compute the derivatives of y with respect to the independent variables. The figure 9 is the plot of the computation times with respect to the number of independent variables. In figure 9, the method using expression templates is clearly the
fastest until the number of independent variables is greater than 50. But it is out the scope of forward mode use.

5.2 Compressible Navier-Stokes and k – ε solver

The NSC2KE Fortran code of B. Mohammadi [9] was rewritten in C++ with the help of C. Berthelot. Then, using the previous pattern, we provide an easy way to use the code as a black box without any modification (figure 10); the code can be used in normal or in automatic differentiation mode. Maximum flexibility was achieved.

5.3 Numerical Results

The computational domain is described in figure 11. For numerical tests, we have done three computations to show the levels of drag control:

− a reference computation without injection/suction,
− a computation with constant injection,
− a computation with controlled injection/suction.

We detail the strategy used in the third computation because it is a bit complex. Computations with an iterative solver starting with a initial solution, require several iterations to obtain a converged numerical solution. And each time the control parameters are changed, we need several iterations to recompute the numerical solution. This operation is rather expensive. For this reason, we compute the minimization loop and the iteration loop of the solver. The new strategy consists in 4 optimization iterations (fixed step gradient) per time iteration and then the numerical solution is stabilized before the next time iteration. As shown in figure 12, this strategy is relevant.

In figure 12, we plot the drag history in the third case. The upper plot is the case without control. The middle one is the case with constant jets, that already minimize the drag. The lower one is the case with active jets, that produce the minimal drag of our strategy.

The numerical method, to compute wall laws, requires a finer mesh closed to the wall. Moreover, in this region, the mesh has to be symmetric with respect to the (xz) axe (see [10] for more details). In figure 13, we have used homothetic circles to ensure a uniform and symmetric mesh around the cylinder.
This produces a mesh composed of 13035 nodes and 25842 triangles.

We plot, in the figure 14, the mach iso-values for the case with active jets.

6 Conclusion

Following the case studied by B. Mohammadi [10], we have proved the usability of the automatic differentiation in C++.

We also provide a flexible and efficient scheme to use (templates) and to do automatic differentiation in forward mode in C++.

But this method requires compilers respecting the latest Standard C++ Draft5. However such compilers are not yet fully implemented and optimized. The result is not always optimal, the compilation process can be time consuming and the compilation requires several hundred of megabytes of memory (200 – 500).

Nevertheless our expression templates scheme improved the performance by a large amount against both ADOL-c and FADBAD, providing a sensible way of using Fad in C++.

A Annex

Listing 16 Specialization (Part I).

/// put a size on each basic type via size0f
template<class T> struct size0f { static const int val = 9999; };

/// gives size0f for particular type
template<> struct size0f<float> { static const int val = 12; };
template<> struct size0f<double> { static const int val = 13; };

/// promote T1 or T2 depending on bool
template<typename T1, typename T2, bool cond> struct promoteNumericalType {};

Listing 17 Specialization (Part II).

/// promote T1 or T2 depending on bool;
///true case return T1

typedef typename T1, typename T2>
struct promoteNumericalType<T1,T2,true>
{ typedef T1 promoted_t; };

/// promote T1 or T2 depending on bool;
///false case return T2
template<typename T1, typename T2>
struct promoteNumericalType<T1,T2,false>
{ typedef T2 promoted_t; };

/// traits class for numerical type promotion
template<typename T1, typename T2>
struct promote{
    /// true if size0f(T1) > size0f(T2)
    enum {
        cond = ( size0f<T1>::val > size0f<T2>::val )
    };

    /// then choose T1 or T2 accordingly
    typedef typenamer
    promoteNumericalType<T1,T2>,
    cond
    >::promoted_t promoted_t;
};

B Annex

(OP,TYPE) are respectively:
- (operator+, FadBinaryAdd),
- (operator-, FadBinaryMinus),
- (operator*, FadBinaryMul) and
- (operator/ FadBinaryDiv).

Listing 18 FadExpr<> OP FadExpr>
template<class E1, class E2>
inline FadExpr<TYPE(FadExpr<E1>, FadExpr<E2>) OP (const FadExpr<E1> &v, const FadExpr<E2> &w) {
    typedef TYPE(FadExpr<E1>, FadExpr<E2>) expr_t;
    return FadExpr<expr_t>(expr_t(v, w));
}

Listing 19 FadCst<> OP Fad<>;
template<class T>
inline FadExpr<TYPE(FadCst<T>, Fad<T>) OP (const A& a, const Fad<T> &e) {
    typedef TYPE(FadCst<T>, Fad<T>) expr_t;
    return FadExpr<expr_t>(expr_t(FadCst<T>(a), e));
}

5 X3J16/96-0225 WG21/N1043
Fig. 12. Drag coefficient values at the last iteration.

Listing 20 frightened<expr> OP FadCst<expr>
template<class E>
inline FadExpr<TYPE<FadExpr<E>>,
   FadCst<typename E::value_type> >
OP(const FadExpr<E> &,
   const typename E::value_type &t) {
   typedef typename E::value_type A;
   typedef TYPE<FadExpr<E>,FadCst<T> > expr_t;
   return FadExpr<expr_t>(expr_t(e,FadCst<T>(t)));
}

Listing 21 Fad<expr> OP FadCst<expr>
template<class T>
inline FadExpr<TYPE<Fad<T>,FadCst<T> > >
OP (const Fad<T> &e, const A &a) {
   typedef TYPE<Fad<T>,FadCst<T> > expr_t;
   return FadExpr<expr_t>(expr_t(e,FadCst<T>(a)));
}

Listing 22 FadCst<expr> OP FadExpr<expr>
template<class E>
inline FadExpr<TYPE<FadCst<typename E::value_type>,
   FadExpr<E> > >
OP (const typename E::value_type &t,
   const FadExpr<E> &e) {
   typedef typename E::value_type A;
   typedef TYPE<FadCst<T>,FadExpr<E> > expr_t;
   return FadExpr<expr_t>(expr_t(FadCst<T>(t),e));
}

Listing 23 FadExpr<expr> OP Fad<expr>
template<class E>
inline FadExpr<TYPE<FadExpr<E>>,
   Fad<typename E::value_type> >
OP( const FadExpr<E> &,
    const typename E::value_type & v) {
   typedef typename E::value_type & v;
   typedef TYPE<FadExpr<E>,
   Fad<typename E::value_type> > expr_t;
   return FadExpr<expr_t>(expr_t(e,v));
}

Listing 24 Fad<expr> OP Fad<expr>
template<class T>
inline FadExpr<TYPE<Fad<T>,Fad<T> > >
OP ( const Fad<T> & e, 
    const Fad<T> & e2 ) {
   typedef TYPE<Fad<T>,Fad<T> > expr_t;
   return FadExpr<expr_t>(expr_t(e1,e2));
}

Listing 25 Fad<expr> OP FadExpr<expr>
template<class E>
inline FadExpr<TYPE<Fad<typename E::value_type>,
   FadExpr<E> > >
OP (const Fad<typename E::value_type> &v, 
    const FadExpr<E> &e) {
   typedef typename E::value_type A;
   typedef TYPE<Fad<typename E::value_type>,
   FadExpr<E> > expr_t;
   return FadExpr<expr_t>(expr_t(v,e));
}
Fig. 13. Cylinder mesh

References

7. Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C : a package for the au-
Fig. 14. Mach iso-values with control.


