Comput Visual Sci 3: 197–208 (2001)

Computing and
Visualization in Science
© Springer-Verlag 2001

Regular article

Automatic differentiation in C++ using expression templates and application to a flow control problem

Pierre Aubert[1], **Nicolas Di Césaré**[2], **Olivier Pironneau**[2]

[1] UMR 5585, Centre de mathématiques, I.N.S.A. Lyon, 69621 Villeurbanne Cedex, France
[2] Laboratoire d'Analyse Numérique, Université Pierre et Marie Curie, Paris VI, 75252 Paris Cedex 05, France

Received: 1 June 1999 / Accepted: 29 May 2000

Communicated by: G. Wittum

Abstract. This work deals with an implementation of automatic differentiation of `C++` computer programs in *forward mode* using operator overloading and *expression templates*. 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. 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.

1 Introduction

Derivatives of functions can be computed exactly not only by hand but also by computers. Commercial softwares such as MAPLE [18] or MATHEMATICA [19] have derivation operators implemented by formal calculus techniques. L. Rall was the first to describe clearly automatic differentiation (AD) in his book [13]. In [7] Griewank presented the `C++` implementation using operators overloading, called ADOL-C. He was inspired by B. Speelpennings' tool JAKE-F that allow insertion of subroutines in order to perform formal calculus on each instruction of a computer program. Thus a function described by its computer implementation can be differentiated *exactly* and *automatically*. This is the reason why it is now called automatic differentiation of computational approximations to functions [6].

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 such a case.

In [11], B. Mohammadi used an 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 another 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 `Fad` (Forward automatic differentiation), so that by simply changing in the source of the flow solver the `C++` reserved word `float` (or `double`) by `Fad<float>` (or `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. Veldhuizen and his `expression templates` [17] technique, an efficient implementation is 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 [11] with the forward mode approach and to report of the efficiency of implementations by `expression templates`.

The paper is organized as follows: in Sect. 2, we recall the concepts of AD and present our implementation in Sect. 3. In Sect. 4, we compare our library with two existing ones. Finally we present the numerical results in Sect. 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 Derivative computation

There are at least three ways to compute derivatives. The first one is analytic, *i.e.* by hand or by using one of the symbolic differentiation packages [18, 19]. 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.

2.1 Automatic differentiation

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

```
**Program1** $P\ (x_1, ..., x_N)$
**for** $k = 1$ to $K$ **do**
    $x_{\mu_k} = \varphi_k\left(x_{D_k}\right)$
**end for**
```

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

```
**Program2** $P\ (x_1, x_2, x_3, x_4, x_5)$
$x_3 = x_2 x_2$
$x_4 = \sin(x_3)$
$x_5 = 2x_1x_3$
$x_5 = x_5/x_4$
```

This produces the following table:

$x_3 = x_2x_2$	$\mu_1 = 3$	$\varphi_1(x) = x^2$	$D_1 = 2$
$x_4 = \sin(x_3)$	$\mu_2 = 4$	$\varphi_2(x) = \sin(x)$	$D_2 = 3$
$x_5 = 2x_1x_3$	$\mu_3 = 5$	$\varphi_3(x, y) = 2xy$	$D_3 = 1, 3$
$x_5 = x_5/x_4$	$\mu_4 = 5$	$\varphi_4(x, y) = x/y$	$D_4 = 5, 4$

2.2 The forward mode

We want to compute the derivatives of P with respect to $(x_i)_{1\le i\le n}$. We denote the gradient of P by $\text{grad}_n P = (\partial_1 P, ..., \partial_n P)^t$. Each x_i has a gradient that we denote by $\text{grad}_n x_i$. Thus the gradient of the k-th instruction of P is given by:

$$\text{grad}_n x_{\mu_k} = \sum_{i\in D_k} \partial_i \varphi_k . \text{grad}_n x_i,\ \ 1 \le k \le K. \tag{1}$$

We compute simultaneously the gradient and the value of each instruction. Before doing computations, we have to set the gradients to an initial value:

$$\text{grad}_n x_i = \begin{cases} e_i & 1 \le i \le n \text{ (independent variables)},\\ 0 & \text{otherwise,}\end{cases}$$

where e_i is the i-th element of the canonical basis.

Finally, the computation of (1) for the last instruction x_{μ_K} will provide us with the gradient of each output variable with respect to $(x_i)_{1\le i\le n}$.

2.3 Implementation of the forward mode

There are two approaches to implement automatic differentiation:

1. A *pre-processing* of the source code that provides a derivated 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. Stauming with FADBAD [2].

A list of currently available tools can be found on the Argonne National Laboratory web site: `http://www.mcs.anl.gov/autodiff/adtools`.

M. Masmoudi and M. Grundman presented us 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 EGCS[1] 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 Fig. 3.

[1] since version 1.0.3, http://egcs.cygnus.com/

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

The pre-processing method used in ADIC is conceptually a better approach for AD even in C++. During the translation process, the program can analyze dependencies on more than one line and then can make use of a better analysis. If you apply an overloading AD tool on a C++ program without doing special things for vector and matrix object, you will do a lot of unused computations because you can't do static dependencies analysis. Thus a partial rewrite of the code is needed in order to get performance back. C++ standard introduce a lot of new features in order to support generic programming: (`expression templates` or `template meta-program`) are good examples of what can be done with such an approach. But building a C++ parser becomes a hard job. Most compilers don't currently support all generic programming features. Thus, we think that an operator overloading approach is simpler and this for several years because it let the burden of parsing C++ program on the compiler.

3 Implementation in C++

3.1 Introduction

Let

$$\mathcal{F}_{arith} = \{+, -, *, /, +(unary), -(unary), ++, --\} \qquad (2)$$

be the set of arithmetic operators,

$$\mathcal{F}_{logical} = \{!=, ==, <, >, <=, =>\} \qquad (3)$$

be the set of logical operators and

$$\mathcal{F}_{math} = \{sqrt, log, pow, ...\} \qquad (4)$$

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 (2), (3) and functions of set defined in (4) available for the classical real types (`double`, `float`) for our `Fad` class.

Listing 3 Forward Automatic Differentiation Class

```
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()`, either 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 ith, const int n)` that set the number of *independent* variables to `n` and set the gradient `dx_` to the `i`-th element of the canonical basis.

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

3.2 *`traits` and automatic return type determination*

`traits` were introduced by N. Myers [12] 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:

```
Fad<TYPE> = Fad<double>
    + Fad<std::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:
  `float` + `double` → `double`,
  `int` + `float` → `float`, ...
- Mathematical rules:
  `double` + `complex` → `complex`, ...
- Mixed rules:
  `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 use 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 structure `returnTypeNotDefined` as the return type.

Listing 4 NumericalTraits, General Definition

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

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

Listing 5 Specialization

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

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

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

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.

3.3 `expression templates` *techniques*

The `expression templates` techniques were introduced by T. Veldhuizen [17] 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

```
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 and loops for `B+C` and `B+C+D` and also copies the results (another loop). This is an internal problem of `C++` that only provides binary operator. `expression templates` allows us to define n-ary operators performing a single loop without temporary.

The basic idea of `expression templates` is to keep this mathematical write (`A=B+C+D`) with the efficiency of the `C` like write:

Listing 8

```
for(int i=0; i<n; ++i)
  A.~dx[i] = B.~dx[i] + C.~dx[i] + D.~dx[i];

A.~val = B.~val + C.~val + D.~val;
```

It is done by building a binary tree composed of elements of $\mathcal{F}_{arith} \cup \mathcal{F}_{logical} \cup \mathcal{F}_{math}$ 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*log(C)` produce the tree of Fig. 1:

[2] http://monet.uwaterloo.ca/blitz

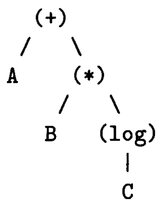

Fig. 1. `expression templates` tree

To build the tree, we define auxiliary classes for all operators in $\mathcal{F}_{arith} \cup \mathcal{F}_{logical}$ and all functions in $\mathcal{F}_{math}$. 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 9, where `value_type` is defined as described in listing 6. That produces the addition operator of listing 10.

Listing 9 Addition Class

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

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

Listing 10 `expression templates` for addition operator

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

The template arguments `L` and `R` are either a class operator like `FadBinary[Add,Minus,Mul,Div]<,>` or `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:

```
FadBinaryAdd<FadBinaryAdd<Fad<T>,
             Fad<T> >,
       Fad<T> >
```

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< >`, taking as argument a variable of type `FadBinaryAdd<,>`, see listing 11. It should be noted that a copy constructor is not needed.

Listing 11 FadBinaryAdd Assignment Operator

```
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 $\mathcal{F}_{arith} \cup \mathcal{F}_{logical} \cup \mathcal{F}_{math}$, *i.e.* for the following classes, `FadBinary[Minus,Mul, Div]<,>` and `FadFunc[Sqrt,Log,Pow,... ]<>` . To clarify the presentation, we show in listing 12 et 13, the pseudo-code of the both implementation, illustrating the Fig. 1.

Listing 12 Pseudo-code of classical approach: `D = A + B * log(C)`

```
//First temporary and first loop : temp1=log(C)
for(i=0; i<n; ++i)
 temp1.dx[i] = C.~dx[i] / C.~val;
temp1.val = log(C.~val);

//Second temporary and second loop :
for(i=0; i<n; ++i)     // temp2=B*temp1
 temp2.dx[i] = B.~val * temp1.dx[i]
         + temp1.val * B.~dx[i];
temp2.val = B.~val * temp1.val;

//Third temporary and third loop :
for(i=0; i<n; ++i)     // temp3=A+temp2
 temp3.dx[i] = A.~dx[i] + temp2.dx[i];
temp3.val = A.~val + temp2.val;

//Assignement and forth loop : D=temp3
for(i=0; i<n; ++i)
  D.~dx[i] = temp3.dx[i];
D.~val = temp3.val;
```

Listing 13 Pseudo-code of `expression templates` approach: `D = A + B * log(C)`

```
//Overloading of val() in class FadFuncLog
temp1_val = log(C.~val);
//Overloading of val() in class FadBinaryMul
temp2_val = B.~val * temp1_val;

for(i=0; i<n; ++i) {
  //Overloading of dx(i) in class FadFuncLog
  temp1_dxi = C.~dx[i] / C.~val;

  //Overloading of dx(i) in class FadBinaryMul
  temp2_dxi = B.~val * temp1_dxi
        + temp1_val * B.~dx[i];

  //Overloading of dx(i) in class FadBinaryAdd
  temp3_dxi = A.~dx[i] + temp2_dxi;

  D.~dx[i] = temp3_dxi;
}

// Overloading of val() in class FadBinaryAdd
D.~val = A.~val + temp2_val;
```

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<>`. 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.

Listing 14 Fad Expression

```
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();}
};
```

The solution of this problem is the definition of a template interface `FadExpr<T>` (see listing 14, in real words an expression of `Fad< >`. It will be the common return type of all operations, where `T` is an operation class like `FadBinaryMinus` or `FadFuncSqrt`. As shown in listing 15, we only change the return type when we define the operator.

Listing 15 `expression templates` addition operator

```
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 ));
}
```

Finally, in listing 16, we only define one assignment operator with `FadExpr<>` class.

Listing 16 Fad Expression Assignment Operator

```
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;
}
```

This kind of operator is called member template operator in the language, and it is not supported by old compilers. This techniques 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.

Listing 17 Fad Constant Class

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

In listing 17, we define the `FadCst` class, where the constructor `FadCst(const T& value)` initialize `constant_` 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`.

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.

3.3.1 A STL like pattern for PDE solver design. There are several ways to use AD in a code. The two first ones are not optimal in the sense of performance but are the simplest. The third one is applied at the beginning of solver development.

In a first time you can just change by query/replace in your editor the keywords `double` or `float` by the corresponding `Fad` classes (`Fad<double>` or `Fad<float>`). But this solution requires to maintain at least two code versions that are a reason of mistakes.

The second solution use `template`. It is slower at first sight but it avoids to maintain several codes. You give the floating type as `template` argument. This requires to modify the whole code but does not require a good understanding of it.

Listing 18

```
// initial version
class Nsc2keSolver {
 Vector<double> solution;
 ...
};

// templatized version
template <class T>
class Nsc2keSolver {
 Vector<T> solution;
 ...
};
```

In these two first methods every variables have the same AD type but are not necessary activate because a variable becomes active when an activate one or an independent one is assigned to it.

The third one is a generic way to implement solvers. We show its flexibility for problems using AD but it can be easily applied to other models.

We can distinguish three kinds of input type for control problems governed by PDE:

- The data type: physical data or problem constants. For example, the study of the position of the shock on an airfoil with respect to the inflow mach number.
- The boundary conditions type for optimal control problems.
- The mesh type for optimal shape design problems.

The solution type of the PDE directly depends on these three types. By example for Navier-Stokes equations, all types are `double`.. Then, the PDE solution type is `double`. Moreover, if you do optimal shape design governed by these equation, the mesh type is `Fad<double>` and the solution type will be `Fad<double>`.

Practically, the idea is to gather all input types in a single class `PdeFactoryTypes`. This is a technique abundantly use in STL classes to provide several type informations. Finally, using numerical traits, we can automatically provide the solution type of the PDE.

Listing 19 Template structure to gather the types

```
template < class Data, class Mesh,
     class Bdy, class Op>
struct PdeTypeFactory {
 typedef Data data_type;
 typedef Mesh mesh_type;
 typedef Bdy boundary_type;

 typedef typename
  NumericalTraits<Data,Mesh>::promote p_T1;
 typedef typename
  NumericalTraits<Bdy,p_T1>::promote pde_type;
};
```

And we define solver classes as template class taking `PdeFactoryTypes` as template argument.

Listing 20 An example of PDE class

```
template <class T> class NSC2KE_Solver {
 T::data_type mach;// Mach number value
 Grid<T::mesh_type> & ReferenceToMesh;
 Vector<T::boundary_type> boundary_values;
 ...
 // the type of solution is a priori unknown,
 // it depends on the problem
 Vector<T::pde_type> solution;
public :
//Solve Navier-Stokes equations
...
};
```

Listing 21 Applications of the previous pattern

```
typedef PdeTypeFactory<double,
           double,
           double> RealPb;

typedef PdeTypeFactory<double,
           double,
           Fad<double> > ControlPb;

// Computation in real mode
NSC2KE_Solver<RealPb> Real_Mode(...);

// Computation in forward mode
NSC2KE_Solver<ControlPb> Control_Mode(...);
```

```
// (AD2)
Fad<double> x1, x2, x3, x4, x5; // ... initialization
x4 = 2.f*(x1*x2)-x3+x3/x5+4.f;

// (AD1)
Fadold<double> y1, y2, y3, y4, y5; // ... initialization
y4 = 2.f*(y1*y2)-y3+y3/y5+4.f;

// (H2)
for (k=0; k<Num; ++k)
  x4.dx(k) = 2.f*(x1.dx(k)*x2.val()+x1.val()*x2.dx(k)) - x3.dx(k)
             + (x5.val()*x3.dx(k)-x3.val()*x5.dx(k))/(x5.val()*x5.val());

x4.val() = 2.f*(x1.val()*x2.val())-x3.val()+x3.val()/x5.val()+4.f

// (H1)
double z1, z2, z3, z4, z5;
double tz1[Num], tz2[Num], tz3[Num], tz4[Num], tz5[Num]; // ... initialization

for (k=0; k<Num; ++k)
  tz4[k] = 2.f*(z2*tz1[k]+z1*tz2[k])-tz3[k]
           + (z5*tz3[k] - z3*tz5[k])/(z5*z5);

z4 = 2.f*(z1*z2)-z3+z3/z5+4.f;
```

Fig. 2. Comparison between differentiation "by hand" (H) and automatic differentiation (AD)

4 Results

4.1 Benchmark

In the first part of this section, we show how the use of `expression templates` is powerful compare to classical overloading. In the second one, we compare our class to ADOL-C 1.8 and FADBAD 2.0 that implement an other technique using overloading.

4.1.1 `expression templates` performances. We have done four computations (Fig. 3) 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 compilers to eliminate temporaries.
  - (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 `nloop` times (to obtain a significant computation times) the expression (Fig. 2),

`2.f*(x1*x2)-x3+x3/x5+4.f`

and it derivatives with respect to the *independent* variables. The five variables are previously activated, this means that their derivatives vectors are allocated to the number of independent variables. The goals of the benchmark are to test the optimization abilities of compilers with the various implementations.

The performances are quite dependent on the compiler. The Figs. 3, 4, 5 and 6 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.

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 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.

[3] version 3.3g, http://www.kai.com/

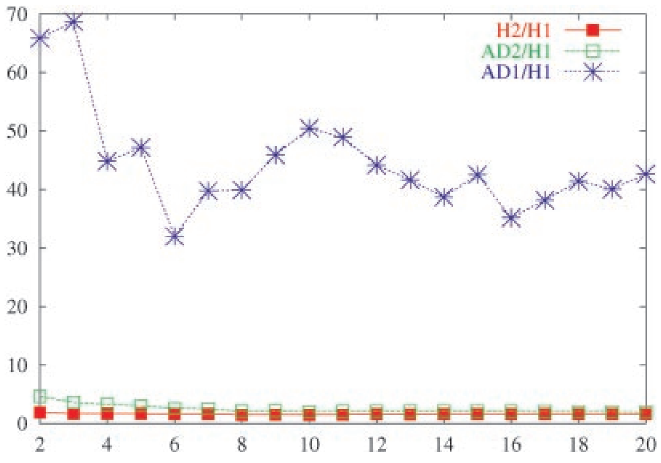

Fig. 3. Computation with EGCS 1.1.2: ratio between the computation times of ($H2$), ($AD1$), ($AD2$) and the reference time ($H1$)

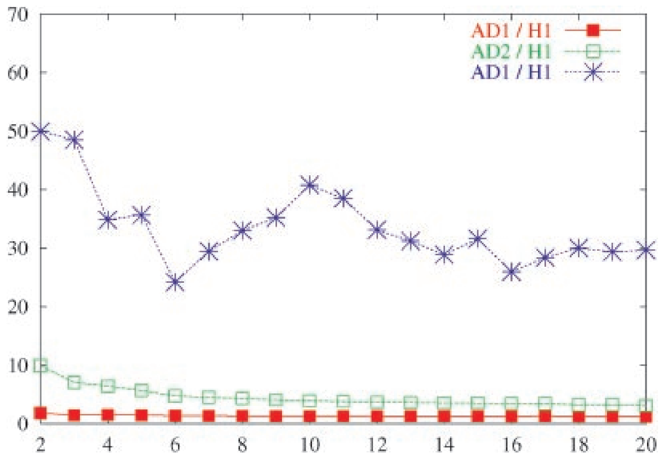

Fig. 4. Same as Fig. 3 with KCC 3.3g

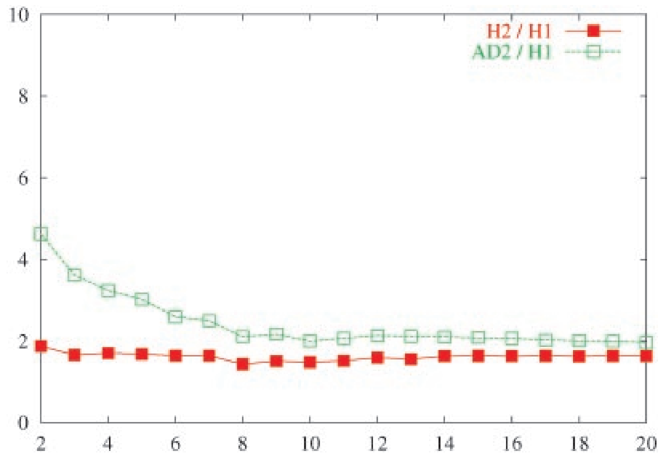

Fig. 5. Zoom of Fig. 3: ratio of computation times for `expression templates` (*AD*2) and member functions (*H*2)

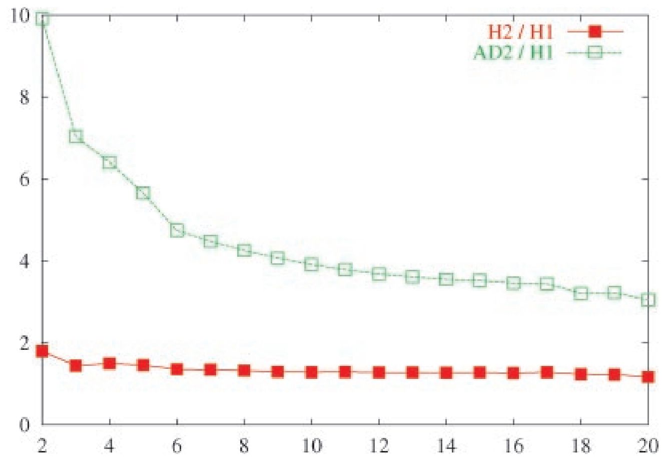

Fig. 6. Zoom of Fig. 4: ratio of computation times for `expression templates` (*AD*2) and member functions (*H*2)

It does not appear on the figures that EGCS is faster that KCC to compute (H1). The ratio H2/H1 is better with KCC (Fig. 6) because it does inlining better than EGCS (Fig. 5).

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 `Nsc2ke++` takes one hour and requires 200 MBytes. 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.[4]).

4.1.2 Comparison with ADOL-C *and* FADBAD. In order to provide a more understandable test, we have also done computations with the forward mode of ADOL-C 1.8 [7] and FADBAD 2.0 [2]. These computations has 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 pointers and not auxiliary template classes. FADBAD uses the classical overloading approach.

Listing 22 ADOL-C, FADBAD and `expression templates` Benchmarks listing

```
// 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

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

// Fad<> 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));
}
```

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 Fig. 7 is the plot of the computation times with respect to the number of *independent* variables. In Fig. 7, the method using `expression templates` is clearly the fastest until the number of *independent* variables is greater than 50. But

[4] http://egcs.cygnus.com/gcse.html

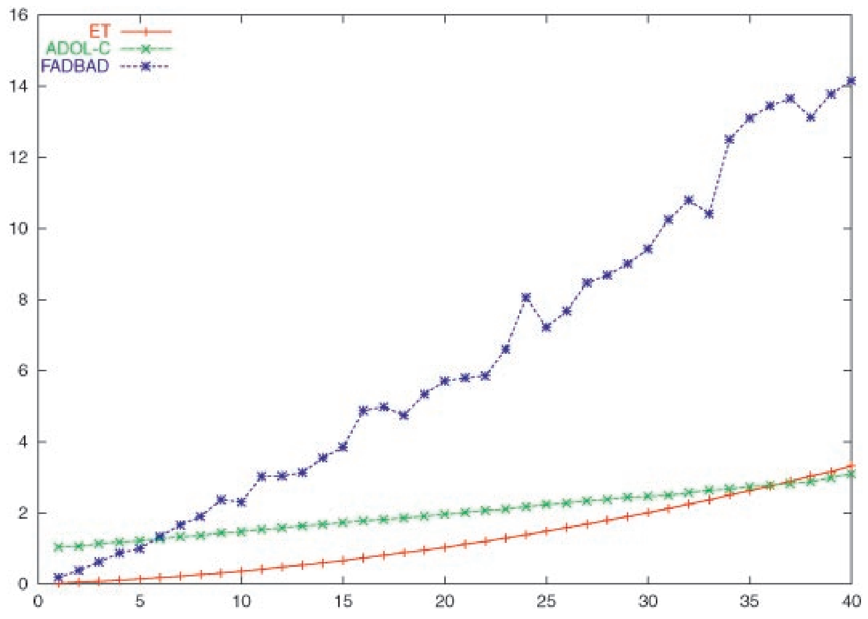

Fig. 7. ADOL-C 1.8, FADBAD 2.0 and the `Fad<>` class comparison

it is out the scope of forward mode use. The `expression templates` has a quadratic grow compare to ADOL-C. It could be explain by the poor inlining capabilities of EGCS. A comparison with KCC should be instructive but ADOL-C does not compile with KCC for the moment.

4.2 Flow Control Problem

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]).

4.2.1 Control law. The flow control behind a cylinder, at Reynolds 42 000 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.

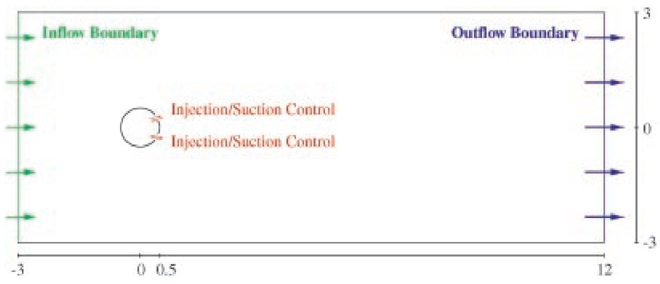

Fig. 8. Flow around a cylinder

There are many ideas to control the unsteadiness of the flow. In this study, we have put the controlled jets behind the cylinder for $30^o \leq |\theta| \leq 40^o$. The boundary condition is defined as an inflow/outflow conditions

$$\int_{\Gamma_{ctrl}} F(W)^t \mathbf{n}\, d\sigma = \int_{\Gamma_{ctrl}} A\, W_{ctrl}(\alpha, \beta)\, d\sigma,$$

where Γ_{ctrl} is the part of the cylinder boundary where the control is applied and α, β are the scalar controls. These scalars are used to control the sign and the norm of the velocity field $\mathbf{u}_{ctrl}$, by means of α and β.

4.2.2 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 (α, β)

$$V(\alpha, \beta) = \min_{\alpha,\beta} J(\alpha, \beta) = \min_{\alpha\,\beta} \int_0^T C_D(\alpha(t), \beta(t)), \tag{5}$$

where

$$C_D(\alpha, \beta) = \frac{1}{2\rho_\infty\, |\mathbf{u}_\infty|^2} \int_{\Gamma_{cyl}} \mathbf{u}_\infty \left(pI - (\mu + \mu_t)\overline{\overline{\mathrm{S}}}\right) \mathbf{n}\, d\sigma,$$

$\mathbf{u}_\infty$ and ρ_∞ are the velocity vector and the density at the inflow boundary, Γ_{cyl} is the cylinder boundary.

4.2.3 Compressible Navier-Stokes and $k-\varepsilon$ solver. The NSC2KE Fortran code of B. Mohammadi [9] was rewritten in `C++` with the help of C. Berthelot. Then, using the solver pattern presented in Sect. 3.3.1, we provide an easy

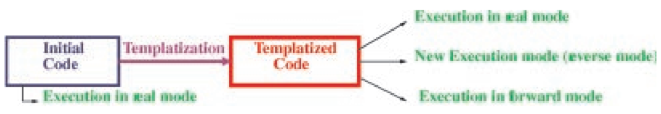

Fig. 9. Template facilities

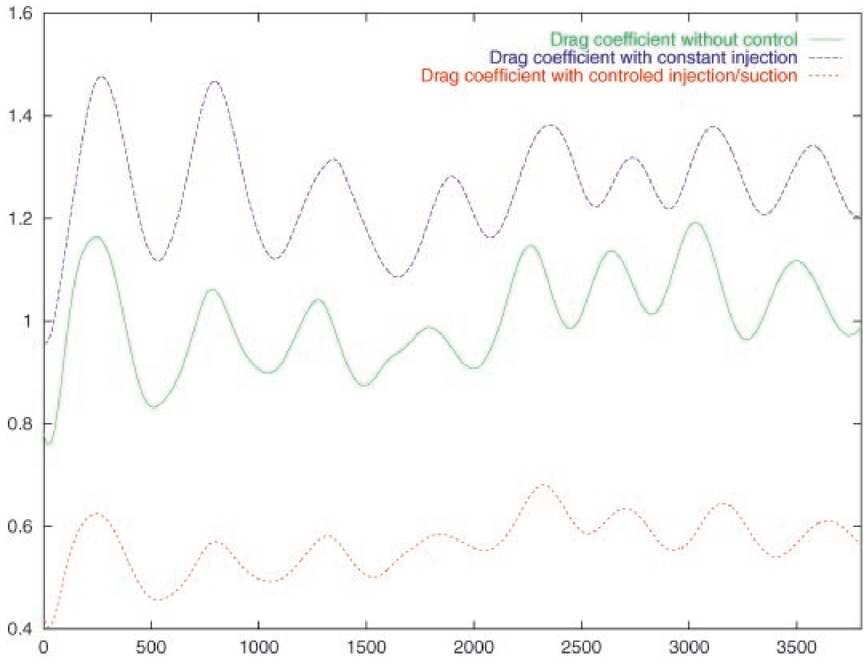

Fig. 10. Drag coefficient values at the last iteration

way to use the code as a black box without any modification (Fig. 9); the code can be used in normal or in automatic differentiation mode. Maximum flexibility was achieved.

4.3 Numerical Results

The computational domain is described in Fig. 8. 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 Fig. 10, this strategy is relevant.

In Fig. 10, 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 (ox) axe (see [10] for more details). In Fig. 11, we have used homothetic circles to ensure a uniform and symmetric mesh around the cylinder. This produces a mesh composed of 13 035 nodes and 25 842 triangles.

We plot, in the Fig. 12, the mach iso-values for the case with active jets.

5 Conclusion

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

We also provide a generic and efficient scheme to enable automatic differentiation in forward mode.

This method requires compilers respecting the latest Standard `C++` Draft[5]. 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 MBytes 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++`.

Annex

Listing 23 Specialization (Part I).

```
/// put a size on each basic type via sizeOf
template<class T>
struct sizeOf {
 static const int val = 9999; };
/// gives SizeOf for particular type
template<> struct sizeOf<float>
 { static const int val = 12; };
template<> struct sizeOf<double>
 { static const int val = 13; };

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

[5] X3J16/96-0225 WG21/N1043

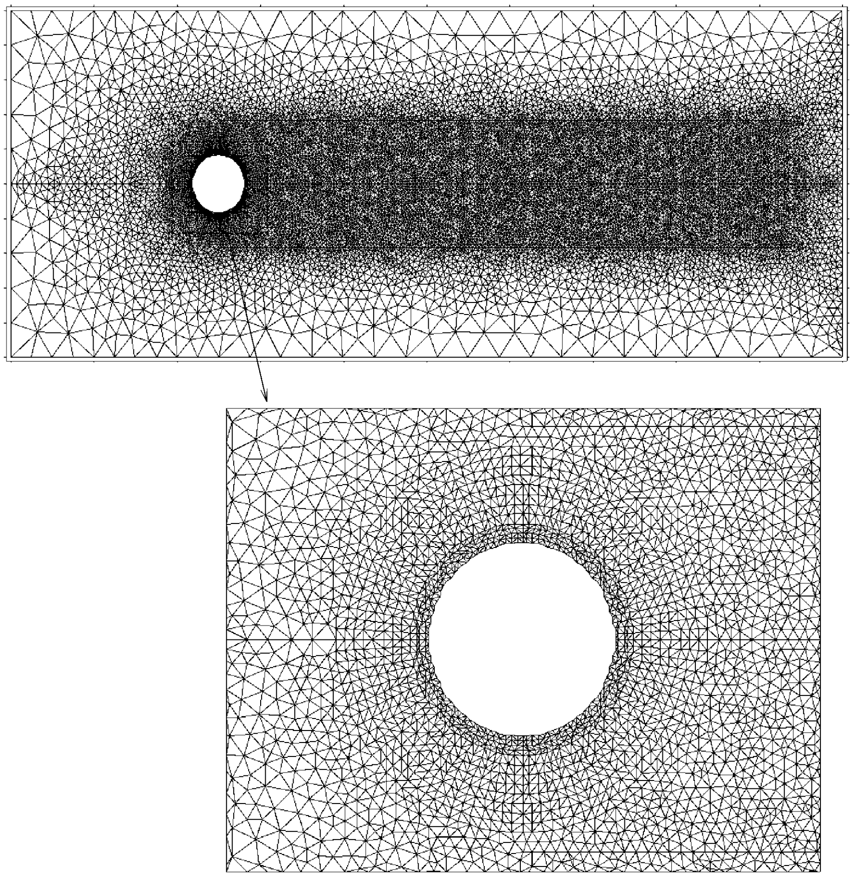

Fig. 11. Cylinder mesh

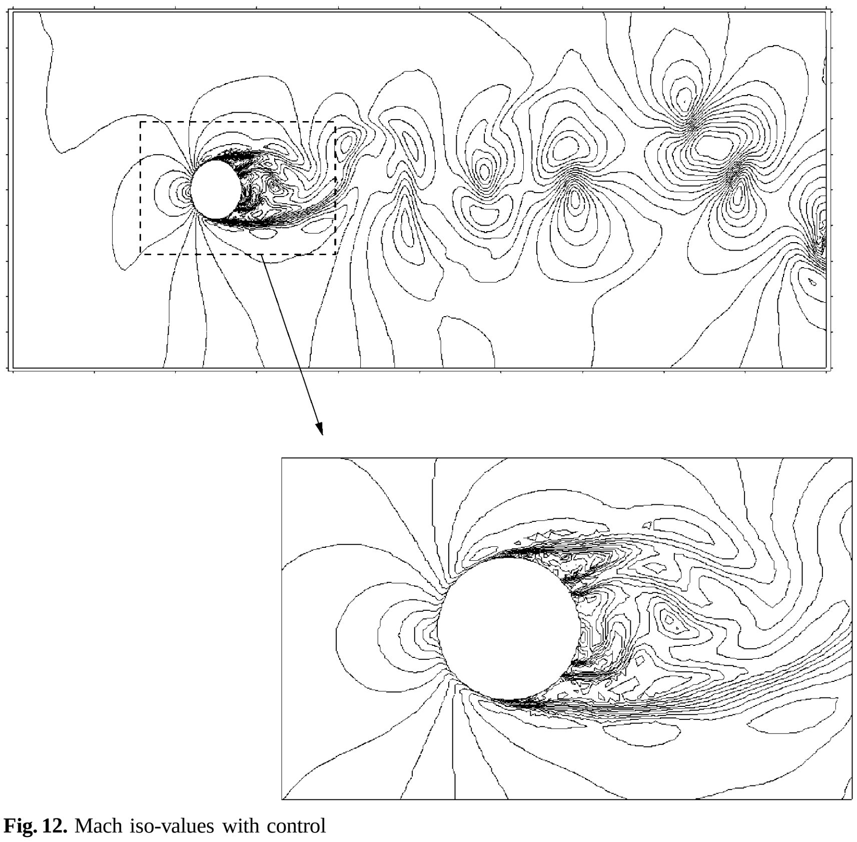

Fig. 12. Mach iso-values with control

Listing 24 Specialization (Part II).

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

/// promote T1 or T2 depending on bool;
///true false 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 sizeOf(T1)> sizeOf(T2)
 enum {
  cond=(sizeOf<T1>::val > sizeOf<T2>::val)
 };
 /// then choose T1 or T2 accordingly
 typedef typename
  promoteNumericalType<T1,T2,cond
         >::promoted_t promoted_t;
};
```

References

1. Barton, J., Nackman, L.: Scientific and Engineering C++. Addison-Wesley 1994
2. Bendtsen, C., Stauning, O.: Fadbad, a flexible C++ package for automatic differentiation using the forward and backward methods. Technical Report IMM-REP-1996-17. 1996
3. Bischof, C., Carle, A., Khademi, P., Mauer, A.: ADIFOR 2.0: Automatic differentiation of fortran 77 programs. IEEE Comp. Science and Engr. 3(3), 18-32 (1996)
4. Bischof, C., Roh, L., Mauer-Oats, A.J.: ADIC: an extensible automatic differentiation tool for ansi-c. Software Practice and Experience. 27(12), 1427-1456 (1997)
5. Desideri, J. A., Dervieux, A.: Compressible flow solvers using unstructured grids. In Computational fluid dynamics. Vol. 1,2. Rhode: Karman Inst. Fluid Dynamics 1988 p. 115
6. Eberhard, P.: Argonne theory institute: Differentiation of computational approximations to functions. SIAM NEWS. 32(1), (1999)
7. Griewank, A., Juedes, D., Utke, J.: Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. j-TOMS. 22(2), 131-167 (1996)
8. Guillaume, P., Masmoudi, M.: Solution to the time-harmonic Maxwell's equations in a waveguide: use of higher-order derivatives for solving the discrete problem. SIAM J. Numer. Anal. 34(4), 1306-1330 (1997)
9. Mohammadi, B.: Fluid dynamics computation with NSC2KE, an user-guide, release 1.0. Technical Report INRIA. (164) 1994
10. Mohammadi, B., Medic, G.: A critical evaluation of the classical $k-\epsilon$ model and wall-laws for unsteady flows over bluff bodies. Int. J. Comput. Fluid Dyn. 10(1), 1-11 (1998)
11. Mohammadi, B.: Contrôle d'instationnarités en couplage fluide-structure. Comptes rendus de l'Académie des sciences – Série IIb – Physique, mécanique, astronomie. 327(1), 115-118 (1999)
12. Myers, N.: A new and useful template technique: "Traits". C++ Report. 7(5), 32-35 (1995)
13. Rall, L.: Automatic Differentiation : Techniques and Applications. Lecture Notes in Computer Science 120. Springer-Verlag 1981
14. Rostaing, N., Dalmas, S., Galligo, A.: Automatic differentiation in Odyssee. Tellus. 45a(5), 558-568 (1993)
15. Stroustrup, B.: The C++ programming language. 3nd edn. Addison-Wesley 1997
16. Veldhuizen, T.: Using C++ template metaprograms. C++ Report. 7(4), 36-43 (1995). Reprinted in C++ Gems, ed. by Stanley Lippman
17. Veldhuizen, T.L.: Expression templates. C++ Report. 7(5), 26-31 (1995). Reprinted in C++ Gems, ed. by Stanley Lippman
18. Waterloo Maple Software. Maple Manual 1995
19. Wolfram Research. Mathematica Manual 1995