Chapter 4
Automatic Differentiation

Contents

1 Introduction **99**

2 Computations of derivatives **99**

2.1 Finite Differences 99
2.2 Complex variables method 100
2.3 State equation linearization 101
2.4 Adjoint method 101
2.5 Adjoint method and Lagrange multipliers 102
2.6 Automatic Differentiation 102
2.6.1 The Direct Mode 102
2.6.2 The Reverse Mode 104
2.7 A Class Library for the Direct Mode 105
2.7.1 Principle of programming 105
2.7.2 Implementation as a C++ Class Library 107

3 Nonlinear PDE and AD **109**

4 A simple inverse problem **112**

5 A shock problem solved by AD **115**

5.1 A simple example $R \to R^2 \to R$ 120
5.1.1 Using the direct mode 121
5.1.2 Using the reverse mode 121
5.1.3 The adjoint code method 121
5.2 DO - IF 122
5.3 Nested Loops 123
5.4 Interprocedural differentiation 128

1 Introduction

This chapter is devoted to the presentation of Automatic Differentiation (AD) of computer programs as a tool for shape optimization.
For gradient based shape optimization methods, it is necessary to have a good evaluation of the derivatives of the discrete cost function with respect to control parameters. It is clear that when the number of control parameters is large, an adjoint equation is necessary [2, 3, 4, 5]. It is tempting to use a discretization of the adjoint equation of the continuous problem, however it would not account for the discretization errors of the numerical schemes (like numerical dissipation for instance). Automatic Differentiation in reverse mode produces the exact derivatives of the discrete cost function. Moreover, the cost of this evaluation is independent of the number of control parameters as for a standard adjoint method.
With this tool, we show first that it can be used also for sensitivity of the numerical scheme itself [9] upon various parameter such as the different numerical fluxes for finite volume methods for compressible inviscid flows. We used Roe and Osher fluxes with and without MUSCL second order reconstruction [10, 13, 11] for the solution of the Euler equations. We showed in [9] that some non differentiable second order schemes are not more precise that first order schemes like the Roe and Osher schemes which are differentiable and hence better suited to AD. Then we show that this approach works equally well on parabolic and hyperbolic equations. Viscous turbulent configurations have also been investigated [1] a $k-\varepsilon$ model [14] with special wall-laws including pressure and convection effects in the Jacobian.
Therefore, AD is not only a tool to evaluate the sensitivities, but it also helps to understand the state equation of a problem and the contribution of each operator involved in the computation of the sensitivities, and this at a reasonable cost.

2 Computations of derivatives

Let us recall four different approaches to find the derivatives of a "cost" function J with respect to control parameters $x \in R^N$, in the case when $J(x, U(x))$ depends on x also via a function $U(x)$ which is the solution of a PDE, the "state equation":

$$J(x, U(x)) \text{ where } x \to U(x) \text{ is defined by } \quad E(x, U(x)) = 0.$$

2.1 Finite Differences

The classical and the most used approach is of course finite differences. Indeed, we need here only multiple cost function evaluations and no additional coding. It is well suited to black-box code users who do not have access at the source code.

$$\frac{dJ}{dx_i} \approx \frac{1}{\epsilon}[J(\vec{x}+\epsilon\vec{e_i}, U(\vec{x}+\epsilon\vec{e_i})) - J(\vec{x}, U(\vec{x}))].$$

However, the two well-known difficulties are :

1. the choice of ϵ (especially for multi-criteria optimizations),
2. the round off error due to the subtraction of nearly equal terms,
3. a computing cost proportional to the size of the state space times the control space.

2.2 Complex variables method

To avoid a critical choice for ϵ and also to avoid the subtraction error present in finite differences, we can use the complex variables method([6, 7]). Indeed as J is real valued we have

$$\frac{dJ}{dx_i} = \frac{Im(J(x_i+\mathbf{i}\epsilon, U(x_i+\mathbf{i}\epsilon)))}{\epsilon} + o(\epsilon),$$

$$J(x_i+\mathbf{i}\epsilon, U(x_i+\mathbf{i}\epsilon)) = J(x, U(x)) + \mathbf{i}\epsilon J'_x - \frac{\epsilon^2}{2}J''_{xx} - \mathbf{i}\frac{\epsilon^3}{6}J'''_{xxx} + o(\epsilon^3),$$

where $x_i + \mathbf{i}\epsilon$ means add to the i_{th} component of the control parameters the increment $\epsilon\sqrt{-1}$. We can see that there is no more subtraction and the choice of ϵ is less critical.
In the same way, the second derivative can be found as:

$$\frac{d^2J}{dx^2} = -2\frac{Re(J(x_i+\mathbf{i}\epsilon, U(x_i+\mathbf{i}\epsilon))) - J(x, U(x))}{\epsilon^2}.$$

Unfortunately, here, there is again a subtraction, but less important than when using a central difference formula ($J''_x = ((J(x+\epsilon) - 2J(x) + J(x-\epsilon))/\epsilon^2)$).
In practice, this methods only requires a redefinition of all real variables of the computer programs in the design loop as complex. This can be seen also as a particular operator overloading approach as in automatic differentiation (see below). On the other hand, the complexity of the approach is comparable to second order finite differences as complex operations and storage requires at least twice the effort compared with reals. Of course, the complexity is still proportional to the number of state times control variables.

2.3 State equation linearization

To reduce the influence of ϵ one way is to use calculus of variation and compute with finite difference only the partial derivatives.
Thus denote $\delta x = \epsilon \vec{e^i}$ and δU the variation of U (i.e. $\delta U = \delta x_i \partial_{x_i} U$). By linearization of $E(x, U(x)) = 0$:

$$\frac{\partial E}{\partial U}\delta U = -\frac{\partial E}{\partial x}\delta x \approx -[E(x+\epsilon\vec{e_i}, U(x)) - E(x, U(x))]. \quad (1)$$

For fluids most numerical implicit solvers are based on a semi-linearization of the equations. For instance Newton's method

$$\frac{\partial E^n}{\partial U}(U^{n+1} - U^n) = -E(x, U^n(x)). \quad (2)$$

In other word, to solve (1), we can use the same implicit solver but with a different right hand side.
More precisely, we have to solve (2) N times the dimension of x, and each solution gives one column of $\partial_x U$.
After substitution, we have:

$$\frac{dJ}{dx} = \frac{\partial J}{\partial x} + \frac{\partial J}{\partial U}\frac{\partial U}{\partial x}.$$

where the partial derivatives are computed by finite differences but $\frac{\partial U}{\partial x}$ is computed by solving (1). But the computing cost problem remains proportional to N times the dimension of U because, as we said, we need to solve N times (1).

2.4 Adjoint method

In the former method we have:

$$\frac{dJ}{dx} = \frac{\partial J}{\partial x} - \frac{\partial J}{\partial U}((\frac{\partial E}{\partial U})^{-1}\frac{\partial E}{\partial x}) = \frac{\partial J}{\partial x} - (\frac{\partial J}{\partial U}(\frac{\partial E}{\partial U})^{-1})\frac{\partial E}{\partial x}.$$

In this expression, we only gathered in a different way the three terms involved in the derivative of the cost function. This simple action is essential, as we can now introduce a new variable p, called the adjoint variable, such that,

$$p^T(\frac{\partial E}{\partial U}) = \frac{\partial J}{\partial U},$$

which enables to compute the gradient at a cost independent of N, as only one solution of (2.4) is required:

$$\frac{dJ}{dx} = \frac{\partial J}{\partial x} - p^T\frac{\partial E}{\partial x}.$$

This can also be linked with a sell-point problem for the Lagrangian.

2.5 Adjoint method and Lagrange multipliers

Introduce the Lagrangian $L = J + p^T E$, and write stationarity with respect to U, p, x. This gives an equation for p:

$$\frac{\partial L}{\partial U} = \frac{\partial J}{\partial U} + p^T \frac{\partial E}{\partial U} = 0, \tag{3}$$

the state equation and:

$$\frac{dJ}{dx} = \frac{\partial L}{\partial x} = \frac{\partial J}{\partial x} + p^T \frac{\partial E}{\partial x}.$$

Notice that (2) is almost similar to (3) and differs only by a transposition of the Jacobian matrix.
Here, the cost is proportional to the state space size plus (and not times) the control space size, because the state and adjoint equations are solved once only; but we still need to compute $\partial E / \partial x$. This can be easily done using finite differences or by AD in direct mode (see below) [19].

This "adjoint method" can be applied at the level of the continuous problem or at the level of the discrete problem. The continuous level has been widely used [2, 3] for various optimization and shape design problems. If we choose to linearize the state equation at the continous level, two difficulties arise,

1. first it does not take into account numerical errors of the discretization
2. and then when the equations are complicated (for instance flow system with turbulence modeling coupled to an elasticity system), it becomes difficult to do the derivation also at the continuous level and produce a bug free code.

The idea is therefore to work at the discrete level and in an automatic fashion.

2.6 Automatic Differentiation

2.6.1 The Direct Mode

The leading principle is that a function given by its computer program can be differentiated by differentiating each lines of the program. So the various methods of AD differs mostly by the way this idea is implemented. A review article on these can be found in [15]. The idea being simple it is best understood on a simple example.

Example: Consider the function J given below and the problem of finding its derivative with respect to u at $u = 2.3$:

$$J'(u) \quad \text{where} \qquad x = 2u(u+1)$$

$$y = x + \sin(u)$$
$$J = x * y$$

Recipee: Above[1] each line of code insert the differentiated line

$$dx = 2 * u * du + 2 * du * (u + 1)$$
$$\mathbf{x = 2 * u * (u + 1)}$$
$$dy = dx + \cos(u) * du$$
$$\mathbf{y = x + \sin(u)}$$
$$dJ = dx * y + x * dy$$
$$\mathbf{J = x * y}$$

The derivative will be obtained by running this program with $u = 2.3, du = 1$,and $dx = dy = 0$ at start time.

Loops and branching statements are not more complicated. Indeed an if statement

```
 A; if ( bool ) then B else C; D;
```

is in fact 2 programs. The first one: A; B; D; gives A';A;B';B; D';D; and the second one A; C; D; gives A';A; C';C; D';D; they can be recombined into

```
A';A; if ( bool) then { B';B} else {C';C} end if; D';D
```

Similarly a loop statement like

```
 A; for i:=1 to 3 do B(i); D;
```

is in fact: A; B(1);B(2);B(3); D; which gives rise to: A';A; B'(1);B(1);...;B'(3);B(3); D';D; which is also recombined into

```
A';A; for i:=1 to 3 do{ B'(i);B(i);} D';D
```

Limitations: However if the variable "bool" and/or if the bounds of the for statement depend on u then there is no way to take that into account. It must be said also that the function is non-differentiable with respect to these variables. There are also some functions which are non differentiable such as $\sqrt{x}$ at $x = 0$. Any attempt to differentiate them at these values will cause an overflow or a NaN (not a number) error message.

When there are several parameters the method remains the same essentially. For instance consider $(u_1, u_2) \to J(u_1, u_2)$ defined by the program

$$y_1 = l_1(u_1, u_2) \quad y_2 = l_2(u_1, u_2, y_1) \quad J = l_3(u_1, u_2, y_1, y_2)$$

[1] because of instructions like x=2*x+1

Apply the same recipe

$$\begin{aligned}
&dy_1 = \partial_{u_1} l_1(u_1,u_2)dx_1 + \partial_{u_2} l_1(u_1,u_2)dx_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1},\mathbf{u_2}) \\
&dy_2 = \partial_{u_1} l_2 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_2 dy_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1},\mathbf{u_2},\mathbf{y_1}) \\
&dJ = \partial_{u_1} l_3 dx_1 + \partial_{u_2} l_2 dx_2 + \partial_{y_1} l_3 dy_1 + \partial_{y_2} l_3 dy_2 \mathbf{J} = \mathbf{l_3}(\mathbf{u_1},\mathbf{u_2},\mathbf{y_1},\mathbf{y_2})
\end{aligned}$$

Run the program twice, first with $dx_1 = 1, dx_2 = 0$, then with $dx_1 = 0, dx_2 = 1$, or, better, duplicate the lines $dy_i =$ and evaluate both at once with $dx_1 = \delta_{ij}$, meaning that

$$\begin{aligned}
&d1y_1 = \partial_{u_1} l_1(u_1,u_2)d1x_1 + \partial_{u_2} l_1(u_1,u_2)d1x_2 \\
&d2y_1 = \partial_{u_1} l_1(u_1,u_2)d2x_1 + \partial_{u_2} l_1(u_1,u_2)d2x_2 \\
&\mathbf{y_1} = \mathbf{l_1}(\mathbf{u_1},\mathbf{u_2}) \\
&d1y_2 = \partial_{u_1} l_2 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_2 d1y_1 \\
&d2y_2 = \partial_{u_1} l_2 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_2 d2y_1 \\
&\mathbf{y_2} = \mathbf{l_2}(\mathbf{u_1},\mathbf{u_2},\mathbf{y_1}) \\
&d1J = \partial_{u_1} l_3 d1x_1 + \partial_{u_2} l_2 d1x_2 + \partial_{y_1} l_3 d1y_1 + \partial_{y_2} l_3 d1y_2 \\
&d2J = \partial_{u_1} l_3 d2x_1 + \partial_{u_2} l_2 d2x_2 + \partial_{y_1} l_3 d2y_1 + \partial_{y_2} l_3 d2y_2 \\
&\mathbf{J} = \mathbf{l_3}(\mathbf{u_1},\mathbf{u_2},\mathbf{y_1},\mathbf{y_2})
\end{aligned}$$

is evaluated with $d1x_1 = 1,\ d1x2 = 0,\ d2x_1 = 0,\ d2x_2 = 1$.

2.6.2 The Reverse Mode

The reference in terms of efficiency for the computation of partial derivatives is the so called "reverse" or adjoint mode.
Consider a simple model problem where J is a function of two parameters and is computed by a program that uses two intermediate variables:

$$\begin{aligned}
&y_1 = l_1(u_1,u_2) \\
&y_2 = l_2(u_1,u_2,y_1) \\
&J = l_3(u_1,u_2,y_1,y_2)
\end{aligned}$$

Let us build the Lagrangian by associating to each intermediate variable a dual or adjoint variable p, except for the last one which is taken equal to 1:

$$L = p_1[y_1 - l_1(u)] + p_2[y_2 - l_2(u,y_1))] + J - l_3(u,y_1,y_2) \tag{4}$$

Stationarity with respect to y_2, y_1 (in that order) gives

$$\begin{aligned} 0 &= p_2 - \frac{\partial l_3}{\partial y_2}(u, y_1, y_2) \\ 0 &= p_1 - p_2 \frac{\partial l_2}{\partial y_1}(u, y_1) - \frac{\partial l_3}{\partial y_1}(u, y_1, y_2) \end{aligned}$$

This gives p_2 first and then p_1 and then J'_u is

$$\frac{\partial J}{\partial u_i} = p_1 \frac{\partial l_1}{\partial u_i} + p_2 \frac{\partial l_2}{\partial u_i} + \frac{\partial l_3}{\partial u_i}$$

The difference with the direct approach is that whatever the number of intermediate variables the adjoint variables p_i are evaluated once only and so the complexity of the computation is much less. On the other hand the method requires a symbolic manipulation of the program itself and so it is not easy to implement as a class library [8].

2.7 A Class Library for the Direct Mode

There are currently several implementations of the reverse mode, Adol-c and Odyssee in particular. But these may not be so easy to use by the beginner: some learning is required.
A very simple implementation of the direct mode can be done in C++ by using operator overloading. It is certainly not a good idea to use this method extensively on problems which have more than 50 control variables or do, but it is so easy and handy that it must be known. It is possible to do the same in FORTRAN 90 as well (see Maikinen[18] for example).

2.7.1 Principle of programming

Consider again

$$J'(u) \quad \text{where} \quad \begin{aligned} & x = 2u(u+1) \\ & y = x + \sin(u) \\ & J = x * y \end{aligned}$$

Each differentiable variable will stores two numbers: its value and the value of its derivative. So we may replace all variables by an array of size 2. Hence the program becomes

```
float y[2],x[2],u[2];
    // dx = 2 u du + 2 du (u+1)
      x[1] = 2 * u[0] * u[1] + 2 * u[1] * (u[0] + 1);
```

```
        // x = 2 u (u+1)
          x[0] = 2 * u[0] * (u[0] + 1);
          y[1] = x[1] + cos(u[0])*u[1];
          y[0] = x[0] + sin(u[0]);
          J[1] = x[1] * y[0] + x[0] * y[1];
        // J = x * y
          J[0] = x[0] * y[0];
```

Now, following [31] we create a C++ class whereby each variable contains the array of size two just introduced and we redefine the standard operations of linear algebra by giving our own definition such as the one used below for the multiplication:

```
#include <iostream.h>

class dfloat{
    public:     float v[2];
    dfloat(){ v[1]=0;} // constructor intialize derivative to 0
    dfloat(double a) { v[0] = a; v[1]=0;}
// promote double into dfloat

    dfloat& operator=(const dfloat& a)
        { v[0] = a.v[0]; v[1] = a.v[1]; return *this;}
 friend dfloat operator * (const dfloat& a, const dfloat& b)
         { dfloat c;
           c.v[1] = a.v[1] * b.v[0] + a.v[0] * b.v[1];
           c.v[0] = a.v[0] * b.v[0];
           return c;
         }
 friend dfloat operator + (const dfloat& a, const dfloat& b)
         { dfloat c;
           c.v[1] = a.v[1] + b.v[1];
           c.v[0] = a.v[0] + b.v[0];
           return c;
         }
// ...
 void init(float x0, float dx0){ v[0]=x0; v[1] = dx0;}
 };

void main () {  dfloat x,u;
   u.init(2.3,1);   // Derivative w/r to u at u=2.3 requested
   x = 2. * u * (u + 1.);
//...
   cout << x.v[0] <<'\t'<< x.v[1] << endl;
```

```
 }
```

This program works as it is (gives 15.18 and 11.2 as expected) but to be complete and applicable to more general programs all the operators like $-, /, >, \sin, \log$... must be added.
The best is to put the class definition into a file "dfloat.h" for instance. Then any C program can be differentiated (FORTRAN also via f2C, but for C++ programs there can be conflicts with other class structures). Now all we have to do is to change all "float" (and/or double) variables into dfloat variables and add a line like "u.init(u0,1)" above to indicate that the derivatives are taken with respect to u at $u = u0$.

2.7.2 Implementation as a C++ Class Library

To handle partial derivatives, the C++ implementation is done with a class construct of the type

```
#include <iostream.h>

template <int N> class dfloat{
    public:     float v[N];
    dfloat()
        { for(int i=1;i<N;i++) v[i]=0;}
    dfloat(double a)
        { v[0] = a; for(int i=1;i<N;i++) v[i]=0;}
    dfloat& operator=(const dfloat& a)
        { for(int i=0;i<N;i++) v[i]=a.v[i]; return *this;}
 friend dfloat operator * (const dfloat& a, const dfloat& b)
         { dfloat c;
           for(int i=1;i<N;i++) c.v[i] = a.v[i] * b.v[0]
                                       + a.v[0] * b.v[i];
           c.v[0] = a.v[0] * b.v[0];
           return c;
         }
 friend dfloat operator + (const dfloat& a, const dfloat& b)
         { dfloat c;
           for(int i=N-1;i>=0;i--) c.v[i] = a.v[i] + b.v[i];
           return c;
         }
// ...
 void init(float x0, int n){ v[0]=x0; v[n] = 1;}
 };

void main () {  dfloat<3> x,u,v;
```

```
   u.init(2.3,1);
   v.init(0.5,2);
   x = 2. * u * ( u + v);
//...
   cout << x.v[0] <<'\t'<< x.v[1]<<'\t'<< x.v[2] << endl;
}
```

In this example the partial derivatives with respect to u and v of $2u(u+v)$ are computed at $u = 2.3, v = 0.5$.

It is better to use a template class because N, which is the number of parameters, should not be fixed at a default value otherwise the library has to be edited by hand each time it is used. Templates are efficient, but here all functions with *for* statements should be implemented outside the class definition for optimal inlining.
As before this program works (answer is 12.88 10.2 4.6) but to make it general all the operation of algebra and all the usual functions like sin() should be added.

In many problems however N may not be known at compile time; this is the case of OSD where is the number of discretization parameters which define the boundary; then dynamic arrays can't be avoided.

```
 class dfloat{
    public: float* v; int N;
    dfloat();
    ~dfloat();
    friend dfloat operator * (dfloat& a, dfloat& b)
         { dfloat c;
           for(int i=1;i<N;i++)
                c.v[i] = a.v[i] * b.v[0] + a.v[0] * b.v[i];
           c.v[0] = a.v[0] * b.v[0];
           return c;
         }
 ...
};
```

where the class dfloat has now a constructor and a destructor to allocate and destroy the array v[].

```
        dfloat::dfloat(){ v = new float[N];
                     for(int i=1;i<N;i++)v[i]=0;} // constructor
        dfloat::~dfloat{ delete [] v;}            // destructor
```

The problem then is that a large number of temporary arrays are created dynamically at execution time, like c.v in the multiplication above, and that takes a lot of computing time.
An optimization can be found in [24] which uses "Expression Templates" and "traits" and which reduces considerably the creation of temporaries so as to arrive at performance similar to those obtained with the template class library explained above. Still to the user the simplicity of the class library is kept and it is extremely easy to link any C program to these libraries for a sensitivity analysis with respect to a few parameters.
P. Aubert [8] made extensive benchmarks between this approach (the forward mode) and the best reverse mode programs; these indicate that this apporach is hard to beat when the number of control parameters is less than 50 or so.
The only possible drawback is that expression templates are rather long and difficult to compile, some compilers can't even do it. The source code of all these classes are available in
`http://www.ann.jussieu.fr/pironnea.`

3 Nonlinear PDE and AD

This is an application of AD to the solution of a nonlinear PDE (here the Navier-Stokes equations in conservation form). We use AD in direct mode for the evaluation of the Jacobian of the flux and use this linearization in a Newton-Gmres algorithm [22].
To solve nonlinear PDE with Newton type algorithm, we need the Jacobian of the operator or its action on a vector. This can be done by linearization or exact computation in the case of differentiable operators, or simply by finite difference (FD). Of course, the two first cases require more programming.
We show here that using AD to provide this Jacobian improve not only the simplicity but also accuracy and robustness and this compared to FD approach. The Newton-GMRES algorithm is used to solve this system without any preconditioning. AD and FD are used to provide an evaluation of the Jacobian. As MUSCL constructions and limiters are used to achieve second order accuracy together with a non differentiable flux (Hybrid Upwind Scheme) [23], an analytic linearization is not possible. More details on the solution of these equations can be found in chapter 3.
We consider the following form of the conservative Navier-Stokes equations:

$$F(W) = \frac{\partial W}{\partial t} + \nabla.(f(W)) = 0, \tag{5}$$

where W is the vector of conservative variables (i.e. $W = (\rho, \rho u, \rho u_2, \rho E)^T$), f represents the advective and viscous operators. This system has 4 equations in

2D for 5 variables and the system is closed using a state law (perfect gas in our case).
W^0 being the initial state, a first order Taylor expansion leads too:

$$F(W^1) = F(W^0)$$

$$+\frac{\partial F}{\partial W}(W^0).(W^1 - W^0) + o((W^1 - W^0)^2).$$

Now, by requiring $F(W^1) = 0$, we get a linearized Newton algorithm:

$$W^1 = W^0 - (\frac{\partial F}{\partial W})^{-1}.F(W^0).$$

We use the GMRES algorithm to solve this system. For the Newton scheme to converge, we need W^0 to be not too far from the solution W^1. But, in our computations, we always start from an uniform state. Therefore, a restart process is used for each computation (this might be seen as a classical time marching).
We present results for a transonic viscous computation around a NACA0012 at Mach 0.85 and Reynolds 1000. Global time stepping has been applied to reduce the mesh effect on the condition of the matrix. The mesh has around 1000 points which makes a problem with 4000 unknowns. The code was run over 350 iterations in time. We recall that no preconditioning has been used for these computations.
We show the numerical values of the different components (4 conservation variables) of the linear tangent Jacobian (i.e. Jacobian times an increment $((\frac{\partial f}{\partial W}).\delta W$) of the flux (HUS + MUSCL + viscous terms + boundary conditions) for 100 points in the mesh obtained using FD and ADP. The ε used for FD computations has been taken for the first (6) and second order (7) FD to give the same results. These gradients are computed from the same state with a CFL number of 100. Of course, during the computation they will diverge from each other. This validates the program produced by the automatic differentiation procedure. We can see that the FD approach always produces smoother gradients.

$$\frac{dF}{dW}.\delta W \sim \frac{F(W + \varepsilon(\delta W)) - F(W - \varepsilon(\delta W))}{2\varepsilon}, \tag{6}$$

$$\frac{dF}{dW}.\delta W \sim \frac{F(W + \varepsilon(\delta W)) - F(W)}{\varepsilon}. \tag{7}$$

In the following pictures, we show the convergence of the GMRES algorithm during the computation. The Krylov space dimension is 20. Two CFL numbers of 10^2 and 10^6 have been tried. For the former case only the use of the Jacobian obtained with the automatic differentiation leads to a good convergence. The

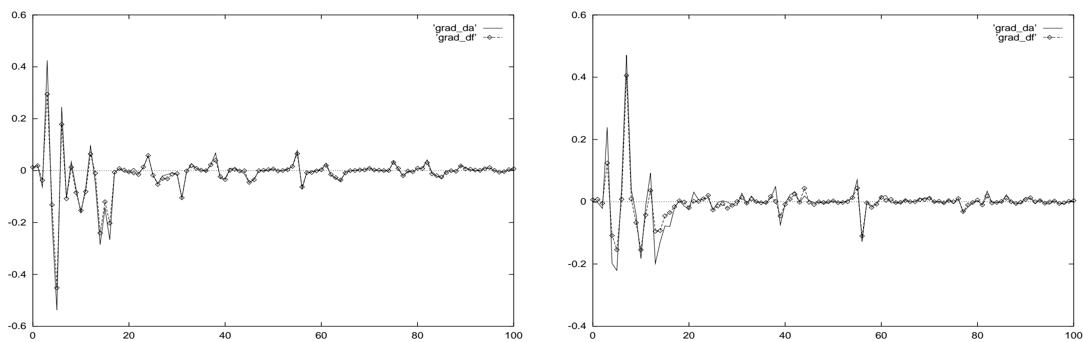


Figure 1: *Linear tangent Jacobian computed by FD and AD for* ρ *and* ρu.

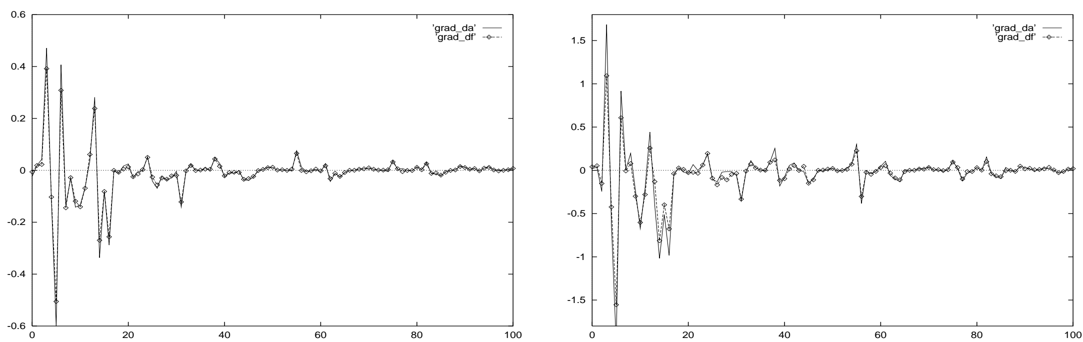


Figure 2: *Linear tangent Jacobian computed by FD and AD for* ρv *and* ρE.

case with $CFL = 10^6$ corresponds to the resolution of Navier-Stokes equations after removing the temporal term. This case can be seen as a resolution with GMRES with 350 (number of time step) restarts.
The convergence history for the previous case and an explicit computation (CFL=1) are shown in picture 4.

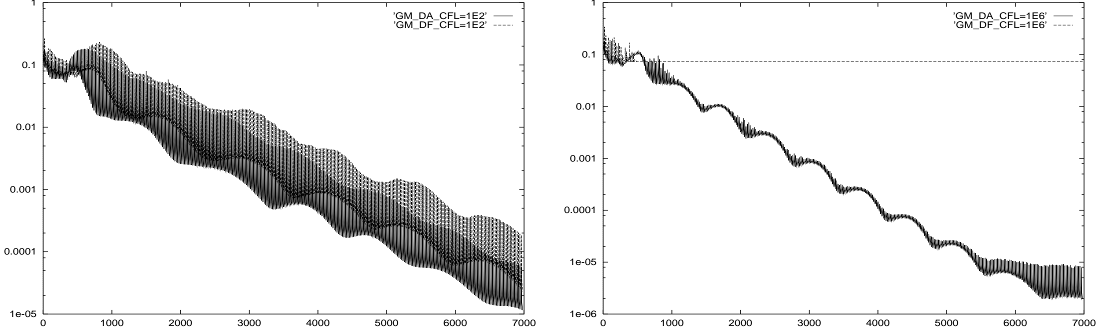


Figure 3: *GMRES convergence for CFL number of* 10^2 *and* 10^6.

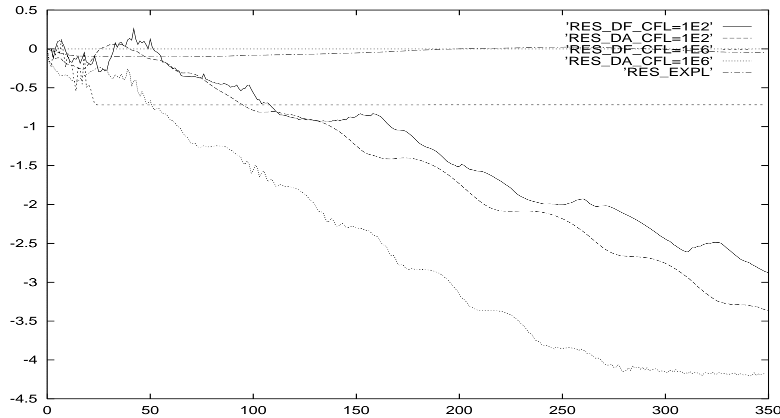


Figure 4: *Global convergence history: explicit, GMRES with the Jacobian using FD and AD. For small CFL numbers* ($CFL = 10^2$) *both the FD and AD approaches converge while for* $CFL = 10^6$ *only ADP works.*

4 A simple inverse problem

We consider here the solution of Burger equation with right hand side. This control problem has been suggested to us by prof. M. Hafez from UC Davis.

$$u(t,x)_t + 0.5(u(t,x)^2)_x = 0.3xu(t,x), \tag{8}$$

$$u(t,-1) = 1, u(t,1) = -0.8, \quad u(0,x) = -0.9x + 0.1.$$

The steady solution of (8) is piecewise parabolic in smooth region and has a jump.

$$u(x) = 0.15x^2 + 0.85 \quad \text{for } x < x_{shock},$$

$$u(x) = 0.15x^2 - 0.95 \quad \text{for } x > x_{shock},$$

and the shock position is found saying that the flux has no jump:

$$u^-_{shock} = -u^+_{shock} \quad \text{therefore} \quad x_{shock} = -\sqrt{1/3}.$$

We use an explicit solver with Roe flux [10] for the discretization. This flux is non differentiable because of the presence of an absolute value. The direct solver for (8) and the sensitivities obtained using direct and reverse mode of AD are shown in appendix. We also show the application of interprocedural differentiation on this example.

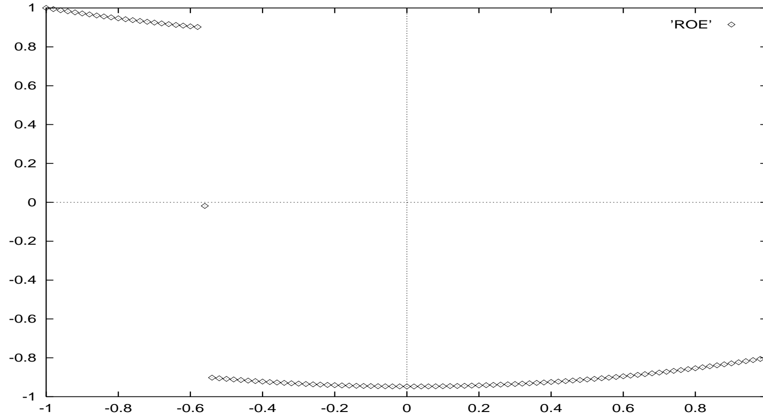


Figure 5: *Solution of the Burger equation with control in the right hand side (discretization with 100 intervals).*

This example shows that such problems have several local minima and that even if the target solution is simple (like linear here), the optimization algorithm can pick one completely different and often more sophisticated. general shape optimization problems based on PDE's solution have the same characteristics.

We want to see how accurate are the derivatives produced by AD compared to finite differences. We choose (6) to evaluate the derivatives for FD. The cost of one evaluation is therefore 101 solutions of the Burger equation.

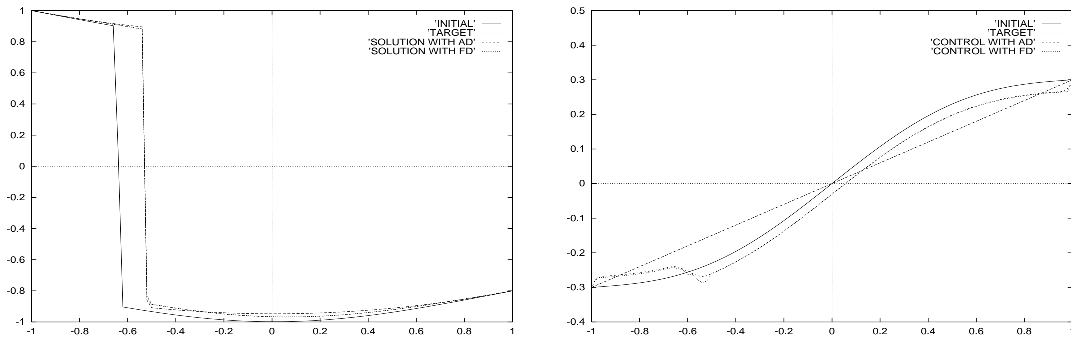


Figure 6: *FD vs. AD : Solution of the Burger equation (left): target, initial and final and the control distribution (right): target, initial and final.*

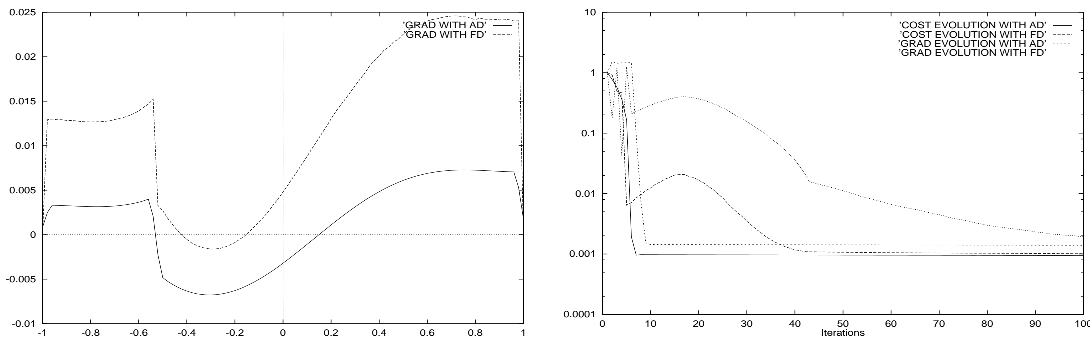


Figure 7: *FD vs. AD : The gradients produced by each approach (left) and the convergence history for the cost and gradient (right). When using automatic differentiation, the convergence is uniform. The final states are however similar.*

We would like to recover this solution by solving the following control problem:

$$u_t + (u^2)_x = F(x)u, \quad u(-1) = 1, u(1) = -0.8, \tag{9}$$

where $F(x) \in \mathbf{R}^{100}$ is the control space, the discretization space has 100 intervals. The cost function is given by:

$$J(x) = \frac{1}{2}\int_{-1}^{1}(u(x) - u_{des}(x))^2 dx, \tag{10}$$

where u_{des} is the solution obtained solving (8) and shown in picture (5). $F(x)$ has been perturbed initially around the target solution. We want to use a gradient method to minimize (10). The descent parameter is set to 0.001. To get the Jacobian of J, we use AD in reverse mode and finite differences.

5 A shock problem solved by AD

To illustrate the power of AD we shall consider the problem of computing the sensitivity of the shock position in a transonic nozzle. Consider the Euler equations :

$$\partial_t W + \nabla \cdot F(W) = 0, \quad W = \begin{pmatrix} \rho \\ \rho\vec{u} \\ \rho E \end{pmatrix}$$

With an entropy condition, initial conditions and boundary conditions. like $W \cdot n|_{in} = g(\alpha)$.

Consider now the problem of computing the derivative V of W with respect to a parameter α apearing in the boundary conditions.
If Calculus of Variation applies we would expect V to be solution of the linearized Euler equations

$$\partial_t V + \nabla \cdot F'(W)V = 0, \quad V = \begin{pmatrix} \rho' \\ (\rho\vec{u})' \\ (\rho E)' \end{pmatrix}$$

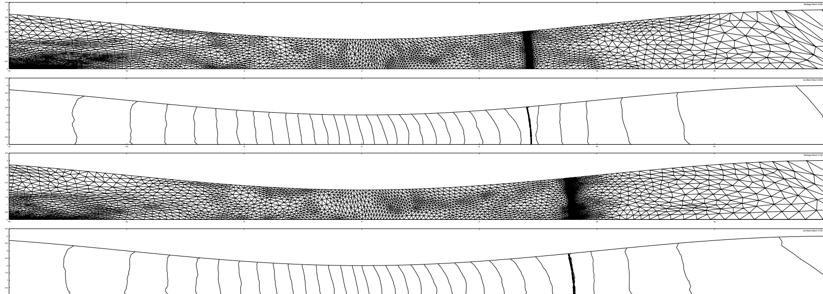
with boundary conditions like $V \cdot n|_{in} = g'(\alpha)$
However if W has a shock we know that the shock position depends upon α.But how can $W + V\delta\alpha$ displace the shock?
Godlewski-Olazabal-Raviart's [28] showed that this is indeed the case for Burgers' equations.

$$\partial_t w + \partial_x f(w) = 0$$

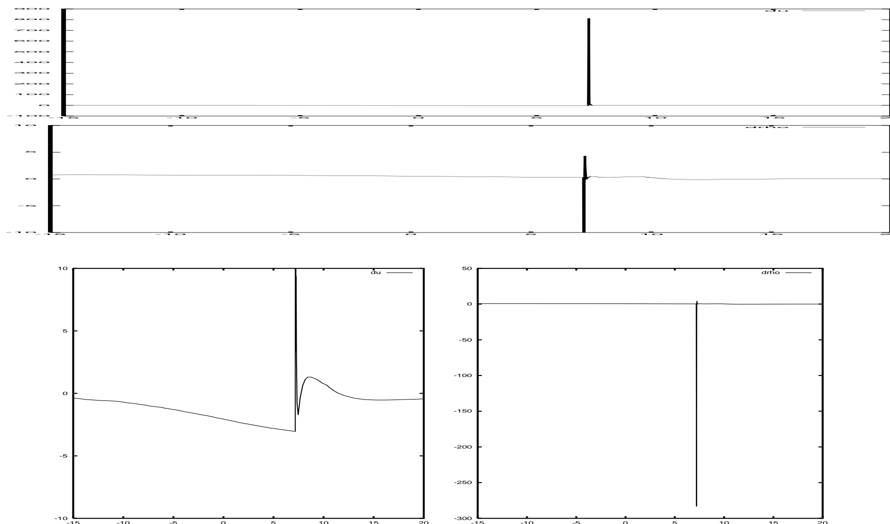
and that the linearized Burgers' equation in weak form

$$\partial_t w, v > + < \partial_x w, f'(u)v >= 0 \ \ \forall w \in C^1(R \times R^+)$$

makes sense provided that $a(u)v$ which is the product of a discontinuous ufnction by a distribution means $[a(u)]\delta_\Sigma$ at the shock position Σ.
The fortran program **nsc2ke** was converted by**f2C**. The C-keyword `float` was been changed into `dfloat`. The line `mach.init(1,1)` has been added and the following results where found.
The first four plots display two computations for two different values of the inlet mach number. Automatic mesh adaption has been used for precise results.



Then the first case was redone with AD. The plots below display the derivative of the mach numberand of the density on the symmetry line (bottom line here) of the nozzle. Then zooms of the same near the shock are shown.



They display a Dirac mass at the shock position .

References

[1] B. Mohammadi (1996),*Optimal Shape Design, Reverse Mode of Automatic Differentiation and Turbulence*, AIAA paper 97-0099.

[2] O. Pironneau (1984),*Optimal Shape Design for Elliptic Systems,* Springer-Verlag, New York.

[3] A. Jameson (1994),*Optimum Aerodynamic Design via Boundary Control*, AGARD Report 803, Von Karman Institute Courses.

[4] O. Baysal, M.E.Eleshaky, (1992) *Aerodynamic Design Optimization using Sensitivity Analysis and CFD*, AIAA Jornal 30(3), pp.718:725, 1992.

[5] M.B.Giles, N.A.Pierce, (1997) *Adjoint Equations in CFD: duality, Boundary Conditions and Solution Behaviour*, AIAA-97-1850.

[6] W. Squire and G. Trapp, (1998), *"Using Complex Variables to Estimate Derivatives of Real Functions"* , siam review, vol. 10, num. 1, pp. 110-112.

[7] K. Anderson and J. Newman and D. Whitfield and E. Nielsen (1999), *"Sensitivity Analysis for the Navier-Stokes Equations on Unstructured Grids using Complex Variables"*, AIAA Paper 99-329.

[8] P. Aubert, (2000) *Cross-section optimisation for curved and twisted 3D beams via a mixed forward and backward optimisation algorithm*, proc. AD 2000 SIAM conference, Nice.

[9] M. Hafez, B. Mohammadi, O. Pironneau (1996),*Optimum Shape Design using Automatic Differentiation in Reverse Mode* , ICNMFD conference, Monterey.

[10] P.L.Roe (1981), *Approximate Riemann Solvers, Parameters Vectors and Difference Schemes,* J.C.P. Vol.43.

[11] G.D.Van Albada, B. Van Leer (1984),*Flux Vector Splitting and Runge-Kutta Methods for the Euler Equations*, ICASE 84-27.

[12] J. Steger, R.F. Warming (1983), *Flux Vector Splitting for the Inviscid gas dynamic with Applications to Finite-Difference Methods*, J. Comp. Phys. 40, pp: 263-293.

[13] S. Osher, S. Chakravarthy (1982), *Upwind Difference Schemes for the Hyperbolic systems of conservation laws*, mathematics of Computation.

[14] B.E. Launder and D.B. Spalding (1972),*Mathematical Models of Turbulence,* Academic Press.

[15] J.C. Gilbert, G. Le Vey, J. Masse (1991),*La différentiation automatique de fonctions représentées par des programmes*, Rapport de Recherche INRIA 1557.

[16] N. Rostaing-Schmidt (1993), *Différentiation automatique: Application à un problème d'optimisation en météorologie*, Ph.D. Thesis, University of Nice.

[17] C. Faure (1996), *Splitting of Algebraic Expressions for Automatic Differentiation,* In proc. of the Second SIAM Inter. Workshop on Computational Differentiation, Santa Fe.

[18] R. Makinen, (1999) *Combining Automatic Derivatives and Hand-coded Derivatives in Sensitivity Analysis for Shape Optimization Problems.* Proc. WCSMO 3, Buffalo, N.Y.

[19] B. Mohammadi, J.M.Malé, N. Rostaing Schmidt (1995), *Automatic Differentiation in Direct and Reverse Modes: Application to Optimum Shapes Design in Fluid Mechanics*, proc. SIAM workshop on AD, Santa Fe, SIAM.

[20] A. Griewank (1995),*Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation,* Optimization Methods and Software, Vol. 1, pp. 35-54.

[21] B. Mohammadi (1996),*Différentiation Automatique par Programme et Optimisation de Formes Aérodynamiques*, MATAPLI 07/96.

[22] P. N. Brown, Y. Saad (1990), *Hybrid Krylov Methods for Nonlinear Systems of Equations,* SIAM J. Sci. Stat. Comp., Vol. 11, No. 3, pp. 450-481.

[23] F. Coquel (1994), The Hybrid Upwind Scheme, private communication.

[24] P. Aubert, N. Dicesare, O. Pironneau, (1999) *Automatic Differentiation in C++ using Expression Templates and Application to a Flow Control Problem.* Submitted to Computer Visualization and Software, Springer.

[25] B. Mohammadi (1994), *Fluid Dynamics Computation with NSC2KE: An User-Guide*, INRIA technical report No. 164.

[26] John Barton and Lee Nackman, (1994) *Scientific and Engineering C++.* Addison-Wesley.

[27] C. Bendtsen and O. Stauning, (1996) *Fadbad, a fexible c++ package for automatic differentiation using the forward and backward methods.* Technical Report IMM-REP-1996-17.

[28] E. Godlewski, M. Olazabal, P.A. Raviart, (1998) *On the Linearization of Hyperbolic Systems of Conservation Laws* , Eq. aux dérivées partielles et applications, pp. 549-570, Gauthier-Villars, proc. colloque en l'honneur de J. L. Lions, Paris.

[29] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. (1992) *ADIFOR : Generating derivative codes from fortran programs.* Scientic Programming, 1(1):11:29.

[30] Peter Eberhard, (1999) *Argonne theory institute : Differentiation of computational approximations to functions.* SIAM NEWS, 32(1).

[31] Ph. Guillaume and M. Masmoudi, (1997) *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):13061330.

[32] Andreas Griewank, David Juedes, and Jean Utke. *Algorithm 755: ADOL-C : a package for the automatic differentiation of algorithms written in C/ C++.* j-TOMS, 22(2):131:167, June 1996.

[33] Bjarne Stroustrup.*The C++ Programming Language.* Addison-Wesley, 3nd edition, 1997.

[34] . Nathan Myers. (1995) *A new and useful template technique: Traits".* C++ Report, 7(5):32:35.

[35] N. Rostaing, S. Dalmas, and A. Galligo. (1993) *Automatic differentiation in Odyssee.* Tellus, 45a(5):558:568

[36] Todd L. Veldhuizen. (1995) *Expression templates.* C++ Report, 7(5):26 31, Reprinted in C++ Gems, ed. Stanley Lippman.

[37] P.L. Viollet. (1981) *On the modeling of turbulent heat and mass transfers for computation of buyoancy affected ows.* Proc. Int. Num. Meth. for Laminar and Turbulent Flows, Venezia.

[38] Waterloo Maple Software. Maple Manual, 1995.

[39] Wolfram Research. Mathematica Manual, 1995.

Appendix 1: Odyssee's Automatic Differentiator

The program used here for computing the gradient of the cost function has been obtained using the automatic differentiator Odyssee developed at INRIA-Sophia Antipolis by the SAFIR team [15, 16, 17, 20].
Odyssee is written using the programming language CAML; it reads standard Fortran 77 programs but some limitations still remain. For instance no *go to line* can be used in reverse mode. One important feature is that Odyssee can treat non-differentiable predefined functions like *min, max, sign, abs, ...* and also perform interprocedural derivation.
Two differentiation procedures are available in Odyssee called the forward and the reverse modes. The forward mode consists of computing the function and its derivative at the same time. This can be seen as the propagation of the informations on the function and its derivatives through the program. We will describe these modes in the next appendix. When using the direct mode, the user has to choose between three algorithms depending on the nature of the performance aimed:
1) Compute the function and all the partial derivatives at the same time. This is the most memory consuming choice.

2) Compute the function and one partial derivative each time. This is quite like using finite differences. This choice is the most time consuming because of the redundant evaluations of the function.
3) Compute the function and save the computational graph of the function, then compute the partial derivatives using the dependency informations from the graph. This choice might be quite memory consuming if the graph is complex.
An estimation of the time needed by the second algorithm for programs where only arithmetic operations are involved is given by:

$$T(f, f') \leq 4nT(f),$$

where $T(f)$ is the time for an evaluation of the function and n is the number of control points. We can see that this is more than for finite differences.
The backward mode can be seen as the adjoint method used in optimization problems for gradient computations. The most important advantage of this approach compared over the previous one is that $T(f, f')/T(f)$ is bounded by a constant independent of the number of control points.

Appendix 2 : Direct and reverse modes of AD

Let, f be a composed function given by:

$$x \in R^p \rightarrow y = h(x) \in R^n \rightarrow z = g(y) \in R^n \rightarrow u = f(z) \in R^q.$$

Following the composed function differentiation rule we have:

$$u' = f'(z)g'(y)h'(x), \tag{11}$$

where $f' \in R^{q\times n}$, $g' \in R^{n\times n}$, $h' \in R^{n\times p}$. We notice that from a practical point of view, in (11) we need to introduce a intermediate matrix $M = g'(y)h'(x) \in R^{p\times n}$ to store the intermediate result before making $u' = f'(z)M$.
Now, after transposition of (11) we have:

$$u'^T = h'^T(x)g'^T(y)f'^T(z). \tag{12}$$

We can see that the storage now is $M = g'^T(y)f'^T(z) \in R^{n\times q}$. It is easy to understand that following the dimensions of the different spaces (i.e. p and q), we should use formula (11) or (12) to optimize the required memory. For instance, for optimization applications where p is the number of control variables and $q = 1$ with f being a cost function, the differentiation after transposition is more suitable.
We call the choice (11) the direct and (12) the reverse mode of differentiation.
In a computer program, the situation is the same. In the direct mode, the Jacobian is produced by differentiating the program (considered as composed function) line by line. The reverse mode is less intuitive, it corresponds to writing the adjoint code (instructions of the direct code in the reverse order). We will describe these modes through the following examples.

5.1 A simple example $R \rightarrow R^2 \rightarrow R$

We give two simple examples of the automatic differentiation in direct and reverse modes. Details of these techniques can be found in [15, 16, 17].
Consider the following function $f = x^2 + 3x$ ($f' = 2x + 3$), written as a composed function:

```
y_1=x
y_2=x**2+2*y_1
f  =y_1+y_2
```

We are looking for the derivative of f with respect of x.

5.1.1 Using the direct mode

A line by line derivation with respect of x will give:

$$\frac{dy_1}{dx} = 1, \quad \frac{dy_2}{dx} = 2x + 2\frac{dy_1}{dx},$$
$$\frac{df}{dx} = \frac{dy_1}{dx} + \frac{dy_2}{dx} = 1 + 2x + 2.$$

We see that we have to store all the intermediate computation before making the final addition.

5.1.2 Using the reverse mode

Here, we consider f as a cost function and the lines of the program as constraints. Hence, we can define an augmented Lagrangian for this program associating one Lagrange multiplier to each affectation:

$$L = y_1 + y_2 + p_1(y_1 - x) + p_2(y_2 - x^2 - 2y_1).$$

The Jacobian is a saddle-point for this Lagrangian. On the other word, the derivative of the Lagrangian with respect to intermediate variables is zero and with respect of the independent variable is the Jacobian of f:

$$\frac{df}{dx} = \frac{\partial L}{\partial x} = -p_1 - 2p_2 x,$$
$$\frac{\partial L}{\partial y_1} = 1 + p_1 - 2p_2 = 0,$$
$$\frac{\partial L}{\partial y_2} = 1 + p_2 = 0.$$

We notice that to get the Jacobian df/dx, the previous equations have to be solved in reverse order through an upper triangular system. We always have an upper triangular system due to the fact that in an affectation we have only one entity in the left hand side. This presentation of the reverse mode is quite elegant but not easy to implement. In Odyssee differentiator tool the adjoint code method, presented below, has been used

5.1.3 The adjoint code method

The idea is two line by line write the adjoint of the direct code taken in reverse order.
The key is that for each affectation $y = y + f(x)$, the dual expression is $p_x = p_x + f' p_y$ with p_x and p_y the dual variables associated to x and y. The previous example becomes:

$$p_{y_1} = p_{y_2} = p_x = 0, \quad p_f = 1,$$
$$p_{y_1} = p_{y_1} + p_f = 1, \quad p_{y_2} = p_{y_2} + p_f = 1,$$
$$p_{y_1} = p_{y_1} + 2p_{y_2} = 3, \quad p_x = p_x + 2xp_{y_2} = 2x,$$
$$p_x = p_x + p_{y_1} = 2x + 3.$$

We can see that no triangular system is solved or stored and that the Lagrangian has not been formed. But we introduced a dual variable for each variable appearing in the right hand side of an affectation.

5.2 DO - IF

The most widely used instructions in finite element, volume or difference solvers are loops and conditional operations (often hidden through abs, min, max, ...).
Consider the evolution of $|u(t)|$ (non-differentiable) with respect of the initial condition u_0. u is solution of:

$$\frac{du}{dt} = -au, \quad u(0) = u_0.$$

We use an explicit discretization:

$$\frac{u^{i+1} - u^i}{\Delta t} = -au^i,$$

which can be programmed as:

$$u = u_0$$
$$do \;\; i = 1, .., N$$
$$v = -au$$
$$u = u + \Delta \; t \; v$$
$$enddo$$
$$f = |u|$$

After expansion:

$$u_1 = u_0, v_1 = -au_1, u_2 = u_1 + \Delta t v_1,$$
$$v_2 = -au_2, ..., v_N = -au_{N-1}, u_{N+1} = u_N + \Delta t v_N,$$

we introduce the Lagrangian of the program as before:

$$L = |u_{N+1}| + p_0(u_1 - u_0) + \sum_{i=1}^{N}(p_i(v_i + au_i) + p'_i(u_{i+1} - u_i + \Delta t v_i)).$$

Optimality conditions give:

$$\frac{\partial L}{\partial u_0} = \frac{\partial f}{\partial u_0} = -p_0,$$
$$\frac{\partial L}{\partial u_1} = p_0 + p_1 a - p'_1,$$
$$\frac{\partial L}{\partial v_i} = p_i + p'_i \Delta t, \quad i = 1, .., N$$
$$\frac{\partial L}{\partial u_i} = p_i a - p'_i, \quad i = 1, .., N$$
$$if(u < 0) \;\; \frac{\partial L}{\partial u_{N+1}} = -1 + p'_N,$$
$$if(u \geq 0) \;\; \frac{\partial L}{\partial u_{N+1}} = 1 + p'_N.$$

The limit of the method is the memory required to store p_i et p'_i, especially if internal loops are present. We can see that the branches of conditional statements are treated separately and that the results are assembled after derivation.

Limitations

There are a few limitations when using the reverse mode of Odyssee. The more important is that goto instructions should not be used (neither RETURN or ENTRY). This is logical as in the reverse mode we have to follow the graph of the program in reverse order and using a goto makes the graph of the program much more complicated.

5.3 Nested Loops

A finite element, difference or volume solver often has nested loops. The Burger solver saw previously has the such structure.

```
subroutine burger_with_roe
!
include 'parameter_definition'
!
! time step loop
!
do kt=1,ktmax
    do i=1,n
    flux(i)=0
    enddo
!
    do i=1,n-1
! centred value
        uc=0.5*(u(i+1)+u(i))
! centred flux
        fl=0.25*(u(i)**2+u(i+1)**2)
! roe flux
        flr=0.5*abs(uc)*(u(i+1)-u(i))
! control
    ctrl=0.25*(contr(i)+contr(i+1))*uc*h
!
        flux(i)  =flux(i)  -fl+flr+ctrl
        flux(i+1)=flux(i+1)+fl-flr+ctrl
        enddo
!
! time marching
!
        do i=2,n-1
        u(i)=u(i)+dt*flux(i)/h
        enddo
!
! boundary conditions
!
        u(1)=ul
        u(n)=ur
enddo
!
! cost function evaluation
!
cost=0.
do i=1,n
cost=cost+0.5*(u(i)-udes(i))**2
enddo

end subroutine burger_with_roe
```

To get `dJ / d contr` . we apply the automatic differentiator Odyssee to this program in direct and reverse modes.
In direct linear tangent mode which means directional differentiation, we get:

```
subroutine burger_with_roe_linear_tangent
!
include 'parameter_definition'
!
! extra saving for AD in direct mode
!
real :: sr01s
integer :: odyn
parameter (odyn = n)
dimension :: contrttl(n)
dimension :: uttl(n)
dimension :: fluxttl(odyn)

  do kt = 1, ktmax
     do i = 1, n
        fluxttl(i) = 0.
        flux(i) = 0.
     end do
     do i = 1, n-1
        ucttl = 0.5*(uttl(i+1)+uttl(i))
        uc = 0.5*(u(i+1)+u(i))
        flttl = 0.25*(2*uttl(i)*u(i)+2*uttl(i+1)*u(i+1))
        fl = 0.25*(u(i)**2+u(i+1)**2)
        if (uc.ge.0.) then
           sr01sttl = ucttl
           sr01s = uc
        else
           sr01sttl = -ucttl
           sr01s = -uc
        end if
        flrttl = 0.5*((uttl(i+1)-uttl(i))*sr01s+(u(i+1)-u(i))*sr01sttl)
        flr = 0.5*sr01s*(u(i+1)-u(i))
        ctrlttl = h*0.25*(ucttl*(contr(i)+contr(i+1))&
    +uc*(contrttl(i)+contrttl(i+1)))
        ctrl = 0.25*(contr(i)+contr(i+1))*uc*h
        fluxttl(i) = fluxttl(i)-flttl+flrttl+ctrlttl
        flux(i) = flux(i)-fl+flr+ctrl
        fluxttl(i+1) = fluxttl(i+1)+flttl-flrttl+ctrlttl
        flux(i+1) = flux(i+1)+fl-flr+ctrl
     end do
     do i = 2, n-1
        uttl(i) = uttl(i)+(dt*fluxttl(i))/h
        u(i) = u(i)+(dt*flux(i))/h
     end do
     uttl(1) = 0.
     u(1) = ul
     uttl(n) = 0.
     u(n) = ur
```

```
  end do
  cost = 0.
  costttl = 0.
  do i = 1, n
     costttl = costttl+0.5*(2*uttl(i)*(u(i)-udes(i)))
     cost = cost+0.5*(u(i)-udes(i))**2
  end do
end subroutine burger_with_roe_linear_tangent
```

We have to run this subroutine for every dimension of the control space and get the corresponding sensitivity in `costttl`.
And in reverse mode, we have:

```
subroutine burger_with_roe_reverse
  !
  include 'parameter_definition'
  !
  !
  ! extra saving for AD in direct mode
  !
  real :: sr01s
  integer :: odyn
  parameter (odyn = n)
  integer :: odyktmax
  parameter (odyktmax = ktmax)
  real :: uccl(n)
  real :: contrccl(n)
  real :: save16(1:odyktmax)
  real :: save15(1:odyktmax)
  real :: save14(2:odyn, 1:odyktmax)
  real :: save5(1:odyn, 1:odyktmax)
  real :: save13(1:odyn, 1:odyktmax)
  logical test7(1:odyn,1:odyktmax)
  real :: save4(1:odyn, 1:odyktmax)
  real :: save12(1:odyn, 1:odyktmax)
  real :: save3(1:odyn, 1:odyktmax)
  real :: save11(1:odyn, 1:odyktmax)
  real :: save2(1:odyn)
  real :: save10(1:odyn, 1:odyktmax)
  real :: save18(1:odyn)
  real :: save9(1:odyn, 1:odyktmax)
  real :: fluxccl(odyn)
  real :: save8(1:odyn, 1:odyktmax)

  !
  ! initializing local variables
  !

  flrccl = 0.
  flccl = 0.
  ctrlccl = 0.
```

```
do n1 = 1, n
   fluxccl(n1) = 0.
end do
sr01sccl = 0.
ucccl = 0.
!
! trajectory
!

save1 = cost
do n1 = 1, n
   save2(n1) = u(n1)
end do
do kt = 1, ktmax
   do i = 1, n
      save3(i,kt) = flux(i)
      flux(i) = 0.
   end do
   do i = 1, n-1
      save4(i,kt) = uc
      uc = 0.5*(u(i+1)+u(i))
      save5(i,kt) = fl
      fl = 0.25*(u(i)**2+u(i+1)**2)
      test7(i,kt) = uc.ge.0.
      if (test7(i,kt)) then
         save8(i,kt) = sr01s
         sr01s = uc
      else
         save9(i,kt) = sr01s
         sr01s = -uc
      end if
      save10(i,kt) = flr
      flr = 0.5*sr01s*(u(i+1)-u(i))
      save11(i,kt) = ctrl
      ctrl = 0.25*(contr(i)+contr(i+1))*uc*h
      save12(i,kt) = flux(i)
      flux(i) = flux(i)-fl+flr+ctrl
      save13(i,kt) = flux(i+1)
      flux(i+1) = flux(i+1)+fl-flr+ctrl
   end do
   do i = 2, n-1
      save14(i,kt) = u(i)
      u(i) = u(i)+(dt*flux(i))/h
   end do
   save15(kt) = u(1)
   u(1) = ul
   save16(kt) = u(n)
   u(n) = ur
end do
save17 = cost
cost = 0.
```

```
do i = 1, n
   save18(i) = cost
   cost = cost+0.5*(u(i)-udes(i))**2
end do
!
! transposed linear forms
!
do i = n, 1, -1
   cost = save18(i)
   uccl(i) = uccl(i)+costccl*((2*(u(i)-udes(i)))*0.5)
end do
costccl = 0.
cost = save17
do kt = ktmax, 1, -1
   u(n) = save16(kt)
   uccl(n) = 0.
   u(1) = save15(kt)
   uccl(1) = 0.
   do i = n-1, 2, -1
      u(i) = save14(i,kt)
      fluxccl(i) = fluxccl(i)+uccl(i)*((1./h)*dt)
   end do
   do i = n-1, 1, -1
      flux(i+1) = save13(i,kt)
      flccl = flccl+fluxccl(i+1)
      flrccl = flrccl-fluxccl(i+1)
      ctrlccl = ctrlccl+fluxccl(i+1)
      flux(i) = save12(i,kt)
      flccl = flccl-fluxccl(i)
      flrccl = flrccl+fluxccl(i)
      ctrlccl = ctrlccl+fluxccl(i)
      ctrl = save11(i,kt)
      ucccl = ucccl+ctrlccl*((contr(i)+contr(i+1))*h*0.25)
      contrccl(i) = contrccl(i)+ctrlccl*(uc*h*0.25)
      contrccl(i+1) = contrccl(i+1)+ctrlccl*(uc*h*0.25)
      ctrlccl = 0.
      flr = save10(i,kt)
      uccl(i+1) = uccl(i+1)+flrccl*(sr01s*0.5)
      uccl(i) = uccl(i)-flrccl*(sr01s*0.5)
      sr01sccl = sr01sccl+flrccl*((u(i+1)-u(i))*0.5)
      flrccl = 0.
      if (test7(i,kt)) then
         sr01s = save8(i,kt)
         ucccl = ucccl+sr01sccl
         sr01sccl = 0.
      else
         sr01s = save9(i,kt)
         ucccl = ucccl-sr01sccl
         sr01sccl = 0.
      end if
      fl = save5(i,kt)
```

```
        uccl(i) = uccl(i)+flccl*((2*u(i))*0.25)
        uccl(i+1) = uccl(i+1)+flccl*((2*u(i+1))*0.25)
        flccl = 0.
        uc = save4(i,kt)
        uccl(i+1) = uccl(i+1)+ucccl*0.5
        uccl(i) = uccl(i)+ucccl*0.5
        ucccl = 0.
     end do
     do i = n, 1, -1
        flux(i) = save3(i,kt)
        fluxccl(i) = 0.
     end do
  end do
  do n1 = n, 1, -1
     u(n1) = save2(n1)
  end do
  cost = save1
end subroutine burger_with_roe_reverse
```

Here, the Jacobian is in `contrccl (n)`. This means that in one run of this subroutine, we get the jacobian.
Rather than to follow the differentiation, it is important to notice that:
* The code produced by the direct mode follows quite well the original code and is therefore easy to read. On the other hand, the code generated by the reverse mode is much more difficult to understand.
* The linear tangent mode does not introduce too much saving but we need (as for finite differences) to run it for each dimension of the control space.
* The reverse mode saves intermediate solutions in arrays of size `(n,ktmax)`. `n` should be seen as the number of nodes in a mesh for a Navier-Stokes computation.

5.4 Interprocedural differentiation

As we said, to avoid the memory difficulty with the reverse mode, we use interprocedural differentiation which consists in replacing the core of an internal loop by a subroutine and to differentiate this subroutine. The previous example becomes:

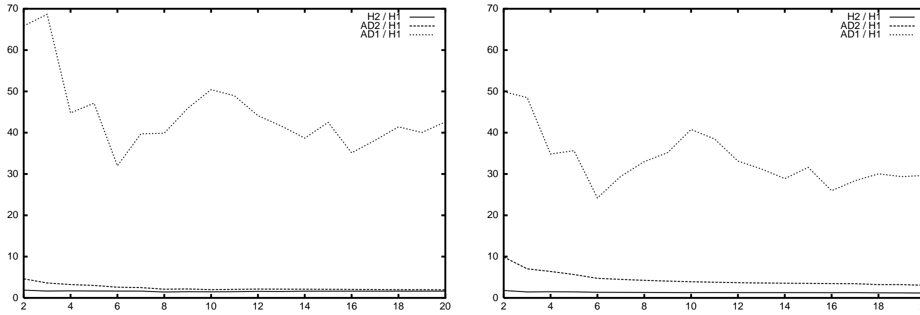
```
subroutine burger_with_roe
!
include 'parameter_definition'
!
! time step loop
!
do kt=1,ktmax
    do i=1,n
    flux(i)=0
    enddo
!
    do i=1,n-1
    call burger_corps (u(i),u(i+1),contr(i),contr(i+1),flux1,flux2)
```

```
        flux(i)  =flux(i)  +flux1
        flux(i+1)=flux(i+1)+flux2
        enddo
!
! time marching
!
        do i=2,n-1
        u(i)=u(i)+dt*flux(i)/h
        enddo
!
! boundary conditions
!
        u(1)=ul
        u(n)=ur
enddo
!
! cost function evaluation
!
cost=0.
do i=1,n
cost=cost+0.5*(u(i)-udes(i))**2
enddo

end subroutine burger_with_roe
!
subroutine burger_corps (u1,u2,contr1,contr2,flux1,flux2)
real :: u1,u2,contr1,contr2,h,dt
! centred value
        uc=0.5*(u2+u1)
! centred flux
        fl=0.25*(u1**2+u2**2)
! roe flux
        flr=0.5*abs(uc)*(u2-u1)
! control
    ctrl=0.25*(contr1+contr2)*uc*h
!
        flux1 = -fl+flr+ctrl
        flux2 = +fl-flr+ctrl
end subroutine burger_corps
```

Appendix 3: More on FAD classes

Derivatives of functions can be computed exactly not only by hand but also by computers. Commercial softwares such as Maple [38] or Mathematica [39] have derivation operators implemented by formal calculus techniques. In [32] 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 [30].



The idea of using operator overloading for AD can be traced in [26], [32] and [27]. It has been used extensively in [31] 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.
Expression templates were introduced by T. Veldhuizen [36] in 1995 for vectors and array computations. Using these techniques, he provided an *active library* called `Blitz++`[2] that enables Fortran performances for C++ programs on several Unix platforms (IBM, HP, SGI, LINUX workstations and Cray T3E).
Expression templates avoid the creation of temporary arrays

- for `y = A*x`, `C++` compilers do `c`$\leftarrow$`A*x`, `c1`$\leftarrow$`c`, `y`$\leftarrow$`c1`

Traits were introduced by N. Myers[34] for 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 example C's promotion rules and mathematical promotion rules can be used in simple cases
C rules : `float` + `double` $\rightarrow$`double`, `int` + `float` $\rightarrow$ `float`, ...
Mathematical rules : `double` + `complex` $\rightarrow$`complex`, ... and we apply the rules to `Fad<>` calculation. But to avoid the creation of a temporary to store the matric vetor product Ax and automatic generator of the type MatrixVectorProductResult must be generated and that is done with traits.

P. Aubert & N. Dicesare wrote `fad.h` which implements and tests these ideas. Comparison Adol-C (Griewank), fadbad 2.0 (Bendtsen-Stauming) and fad(Aubert-Dicesare), and without expression templates and analytic shows that it is equal to analytic by the same method and better by a factor of 10 when the number of variables in larger than say 20 or so.
Performance with EGCS 1.1.2 and with KCC 3.3g: Comparison between by hand (H), by overloading without expression templates and with expression templates. The number variables goes from 1 to 20. We compute `nloop` times `2.f*(x1*x2)-x3+x3/x5+4.f`
In order to provide a more understandable test computations have been done with the forward mode of ADOL-C 1.8 [32] and FADBAD 2.0 [27]. These computations were 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.

Benchmarks listing

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

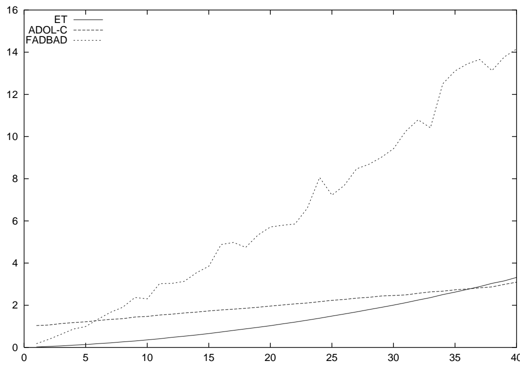


Figure 8: Adol-c 1.8, fadbad 2.0 and the `Fad<>` class comparison.

```
// 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));
}
```

The test defines a vector of *independent* variables and performs several arithmetic operations on this vector that are accumulated in a single variable `y`. The test computes the derivatives of `y` with respect to the *independent* variables. The figure 8 is the plot of the computation times with respect to the number of *independent* variables. In figure 8, the method using Expression Templates (ET) is clearly the fastest until the number of *independent* variables is greater than 50. But it is out the scope of forward mode use. ET has a quadratic grow compare to Adol-c. It could e 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.