

# Automatic Differentiation of a Time-Dependent CFD Solver for Optimisation of a Synthetic Jet

E.M. Tadjouddine<sup>\*1</sup>, S.A.Forth<sup>1</sup>, and N. Qin<sup>2</sup>

<sup>1</sup> Applied Maths & Operational Research Group, Engineering Systems Department, Cranfield University, Shrivenham Campus, SN6 8LA, U.K.

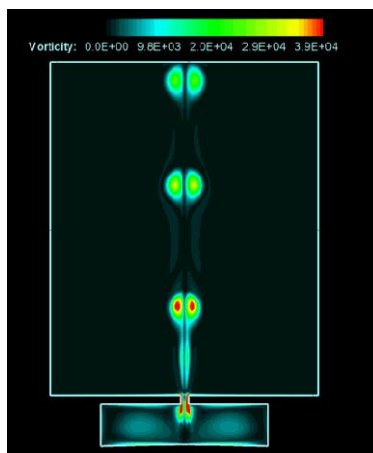
<sup>2</sup> Department of Mechanical Engineering, The University of Sheffield, Mappin Street, S1 3JD, U.K.

Received 29/11/2005

We have used Automatic Differentiation to linearise a time-dependent CFD solver that is written in Fortran 95 and features mesh movement. The CFD code simulates a small device that generates a jet-like flow by oscillating fluid in a chamber connected to its surroundings via an orifice. Using the resulting derivative code, we discuss a case study aimed at optimising the kinetic energy of the upwards motion of the jet by controlling the period and the amplitude of the oscillation.

Copyright line will be provided by the publisher

Synthetic jet actuators are small scale devices generating a jet-like motion by oscillating fluid in a chamber connected to an external flow via an orifice. They may be embedded into an aircraft wing to control flow separation or they can serve as a propulsion system for microfluid systems [10, 21]. The associated fluid mechanics is governed by the unsteady Navier-Stokes equations that are solved using a finite volume approach on a moving mesh [21]. Figure 1 shows the vorticity contours using a two-dimensional geometry, see [21] for further details.



**Fig. 1** Vorticity contours of the synthetic jet flow

In this study, we consider the synthetic jet actuator modelled in [21] and aim to maximise the kinetic energy of the jet in the upwards movement for a given fixed amount of work that cannot be exceeded by the system. The optimisation is carried out using a gradient-based algorithm from the MATLAB optimisation toolbox. Finite-differencing (FD) is a popular way of calculating first-order derivatives. It is easy to implement but incurs truncation errors that may affect robustness of algorithms that use derivatives. An alternative is to use the algorithms and tools of Automatic Differentiation (AD) [19, 12], also known as Algorithmic or Computational Differentiation by which derivatives of a function represented by a computer program are computed without the truncation errors associated with finite-differencing. This facilitates the generation of the gradient of the objective function

\* Corresponding author: e-mail: M.Tadjouddine@cranfield.ac.uk, Phone: +00 44 1793 785889, Fax: +00 44 1793 784196

Copyright line will be provided by the publisher

represented by, in our case, some 4500 lines of Fortran 95 code. Performance of the AD-generated code is enhanced using manual modifications. Preliminary results of an optimisation test case aiming at maximising the kinetic energy of the upstream jet flow using the MATLAB optimisation routine `fmincon` have shown improved solution and convergence as compared to using FD.

## 1 Computing derivatives via Automatic Differentiation

Solving nonlinear systems or nonlinear optimisation often requires derivative computation. Derivatives can be obtained by hand when they are easy. However, if the functions are too complicated, we may use the popular finite differencing scheme, the complex-step derivative approximation [17] or the truncation-error-free derivative evaluation known as Automatic Differentiation [12, 19].

Automatic Differentiation of a computer code representing a function  $\mathbf{F} : \mathbf{R}^n \mapsto \mathbf{R}^m$  can be viewed as a program transformation in which the original code's statements that calculate real valued variables are augmented with additional statements to calculate their derivatives. This is carried out by regarding the original computer code as a sequence of elementary functions having at least a first derivative and then using the chain rule to automatically evaluate the function represented by the given code as well as its derivative. The main advantage of this technique is that it can handle codes of arbitrary complexity to provides accurate and fast derivatives while being reliable compared to hand-coding.

Assuming  $dx$  is the derivative associated with a variable  $x$  and  $x_1, x_2$  the variables with respect to which we desire derivatives, the function  $\mathbf{F} : \mathbf{R}^2 \mapsto \mathbf{R} : (x_1, x_2) \mapsto x_1 x_2 / (1 + x_1^2)$  represented by the following code can be transformed as follows:

$$\begin{array}{rcl}
 v_1 & = & x_1 x_2 \\
 v_2 & = & 1 + x_1^2 \\
 y & = & v_1 / v_2
 \end{array}
 \Rightarrow
 \begin{array}{rcl}
 dv_1 & = & x_2 dx_1 + x_1 dx_2 \\
 v_1 & = & x_1 x_2 \\
 dv_2 & = & 2x_1 dx_1 \\
 v_2 & = & 1 + x_1^2 \\
 dy & = & (v_2 dv_1 - v_1 dv_2) / v_2^2 \\
 y & = & v_1 / v_2
 \end{array}
 \quad (1)$$

In AD terminology, we define the *independent* variables to be those input variables with respect to which we need to compute the derivatives, the *dependent* variables to be those outputs whose derivatives are desired, and the *intermediate* variables to be those whose value depends on an independent and affects a dependent variable. An *active* variable is an independent, intermediate, or dependent variable.

We distinguish at least two standard AD algorithms: the forward mode and the reverse or adjoint mode. The forward mode propagates directional derivatives along the control flow of the original program. The cost of evaluating  $\nabla \mathbf{F}$  (see [9, 12] for more details) is bounded above as follows:

$$\text{Cost}(\nabla \mathbf{F}) \leq 3n \text{Cost}(\mathbf{F}) \quad (2)$$

The adjoint mode is composed of two passes: a forward pass that computes the function and a reverse pass that calculates the sensitivities of the dependent variables with respect to the intermediate and independent variables in the reverse order to their calculation in the function. In the reverse pass, we may need to recompute intermediate values required by the differentiation process or extract them from a storage data structure called the tape (essentially a LIFO stack) if they have been stored during the forward pass. The sensitivities of the dependent to the independent variables give the desired derivatives. The cost of calculating  $\nabla \mathbf{F}$  (see [9, 12] for more details) is bounded above as follows:

$$\text{Cost}(\nabla \mathbf{F}) \leq 3m \text{Cost}(\mathbf{F}) \quad (3)$$

Note that the adjoint mode is particularly efficient in calculating gradients ( $m=1$ ) since its complexity is not dependent on the number of inputs. However, the storage requirement may increase dramatically and powerful strategies to reduce the tape size are required in order to approach the complexity bound shown in (3).

The forward and adjoint modes of AD are implemented in various AD tools that usually use the following approaches:

- Augmenting the given computer code with extra statements calculating derivatives and output a transformed computer code (e.g., ADIFOR 3.0 [2, 3], TAF [6], TAPENADE 2.1 [16]).
- Providing a library that overloads the elementary operations to support derivatives calculation (e.g., ADOLC [14], MAD [8], ADIMAT [1]).

In this paper we are concerned with AD software for numerical codes written in the Fortran programming language. Examples of such AD tools are ADIFOR 3.0, TAF, and TAPENADE 2.1 that implement the two AD modes with variant strategies for performance purposes (see [www.autodiff.org](http://www.autodiff.org) for references and details).

To differentiate a given source code using an AD package, we usually specify the independents, dependents, and the toplevel routine, and then choose an AD algorithm (e.g., forward mode AD). However, the process is frequently not that straightforward for large-scale applications since current AD tools are limited by their language coverage. This limitation often forces the AD user to rewrite his or her input code before being transformed by the chosen software. This phase of code preparation may be aided by scripting languages (e.g., SED, PERL, or PYTHON) to automate the rewrite process. Examples of Fortran features not currently well handled by current AD tools include structured or derived data types, modules, dynamic allocation, pointers, or control structures such as `cycle` or `exit`. Consequently, an input code that uses such features may need to be rewritten prior to differentiation. In this preparation process, the AD user must validate each transformation by checking the semantic of the input code is preserved. This may become a difficult task when the input program is a legacy code that has been validated or developed in a third party location. Eventually, when the input code can be read and analysed by the AD tool, a transformed code that computes the original function and its derivatives will be generated. The obtained computer program need be compiled and run to get derivative values. This leads us to seek ways of checking the correctness of derivative values hence validating the AD generated code and of improving its performance.

## 1.1 Validation

To ensure an AD generated code calculates the correct derivative values, it is important to check that the AD obtained values are in line with those from finite-differencing and that different AD algorithms give the same values to within the limits of the machine precision. We may proceed as follows:

1. Compute a single directional derivative  $\dot{\mathbf{y}} = \nabla F(\mathbf{x})\dot{\mathbf{x}}$  for a random direction  $\dot{\mathbf{x}}$  by using finite-differencing and the forward mode AD. Then check that the difference between the two values is about the square root of the machine precision  $\epsilon$ .
2. Compute a single adjoint  $\bar{\mathbf{x}} = \nabla F(\mathbf{x})^T \bar{\mathbf{y}}$  for a random  $\bar{\mathbf{y}}$  via the reverse mode AD. Then check that  $\bar{\mathbf{y}}\dot{\mathbf{y}} \equiv \bar{\mathbf{x}}\dot{\mathbf{x}}$ .

This validation process is crucial and needs careful attention. Note that AD makes the assumption that the input function  $\mathbf{F}$  to be differentiated is composed of elemental functions  $\phi$  that are continuously differentiable on their open domains [12]. However, this may not be the case for real-life applications. At a point on the boundary of an open domain, the function  $\mathbf{F}$  can be continuous, but its derivative  $\nabla \mathbf{F}$  may jump to a finite value or even infinity. A black box approach of AD can lead to wrong results in the presence of non-differentiable functions or iterative processes.

### 1.1.1 About non-differentiability

A real-life application may contain mathematical functions that are not differentiable in some points in their domain. A computer code that models such an application may contain intrinsic functions (e.g. `abs`, `sqrt`, or `arccos`) or branching constructs used to treat physical constraints for instance non physical values of model parameters. We now describe three situations, which may cause non-differentiability problems.

First, let us consider the case related to non-differentiable intrinsic functions. For instance, the derivative of  $\cos^{-1}$  is not defined at  $x = 0$  since

$$\frac{d \cos^{-1}(x = 1)}{dx} = \infty.$$

Moreover, consider the function `abs`. Its derivative evaluated at the point  $x = 0$  has more than one possible values including  $-1, 0, 1$ . Choosing one of these values depends upon the numerical application. This suggests that there is no “automatic” way of treating such a pathological case and that code insight is crucial in guiding sensible choices. To date, the best thing an AD tool can do is to provide an exception handling mechanism that can be turned on in order to track down intrinsic related non-differentiable points. ADIFOR 3.0 is a primary example for such a mechanism and to our knowledge, at the time of the writing, it is unique in that respect.

Second, let us consider pathological cases related to branching constructs. Differentiating blindly such a construct may give point-valued derivatives. Therefore a function  $\mathbf{F}$  that is mathematically differentiable may become non-differentiable when AD is applied. This happens namely when the test used in the branching construct involves active variables as in the following example:

$$\text{if } x == 0 \text{ then } y = 1.0 \text{ else } y = x + 1.0 \quad (4)$$

An AD tool will give a derivative value zero at  $x=0$  in lieu of the value one. Although in this example, the branching construct is not needed, there may be cases where such constructs may be used to prevent unphysical values. Currently AD tools do not handle or even detect these cases. Tracking down such pathological cases may require developing robust program analysis algorithms or rely on the user’s insight of the given computer code.

Third, consider an engineering application in which the independent or dependent variables are real-valued but complex-valued data have been used for computation purposes. Using the equivalence between  $\mathbf{R}^2$  and  $\mathbf{C}$ , a complex function  $h : a + ib \mapsto f(a, b) + ig(a, b)$  of a complex variable  $a + ib$ , where  $a, b$  are real values and  $f, g$  are real-valued functions, is differentiable if and only if  $h$  is *analytic* meaning  $\frac{\partial f}{\partial a} = \frac{\partial g}{\partial b}$  and  $\frac{\partial f}{\partial b} = -\frac{\partial g}{\partial a}$ . It follows that the conjugate operator  $z \mapsto \bar{z}$  is not differentiable. The application of AD into such complex-valued functions is discussed in [18]. Unlike real-valued functions, complex-valued functions may be many-to-one mappings. For example, the function `sqrt` maps a complex number  $x=a + ib$  to two complex numbers  $z$  and  $-z$  with  $z = \sqrt{1/2(a + \sqrt{a^2 + b^2})} + i\sqrt{1/2(-a + \sqrt{a^2 + b^2})}$ . This may raise subtle issues for the application of AD. It also turns out that the modulus function `abs` raises issues when evaluated at the origin. To handle possible singularities, the application of the chain rule must be robustly used by the AD tools [18].

### 1.1.2 Iterative Numerical Solvers

An important question in using AD concerns differentiating through iterative processes. Typically, AD augments the given iteration with statements calculating derivatives. Empirically, AD provides the desired derivatives. However, questions remained as to whether the AD generated iteration converges and what it converges to. Consider Fischer’s example as discussed in [7]. The iterative constructor  $x_{k+1} = g_k(x_k)$  with

$$g_k(x) = x \exp(-kx^2) \quad (5)$$

converges to  $g \equiv 0$  when  $k \rightarrow \infty$  whilst its derivative  $g'_k(x) \rightarrow 0$  but  $g'(0) = 1$ . The issues of derivative convergence for iterative solvers in relation to AD are discussed in detail in [11, 13] for the forward mode AD and in [4] for the adjoint mode. In [13], it is been shown that the mechanical application of AD to a fixpoint iteration gives a derivative fixpoint iteration that converges R-linearly to the desired derivative for a large class of nicely contractive iterates or secant updating methods.

Usually, current AD tools generate derivative code using the same number of iterations as the original solver. However, if the initial guess is close to the solution, then this adjoint solver does no longer converge to the adjoint of the solution. For example, let us consider the following implicit iterative solver:

$$z_0 = z_0(x, y), \quad z_i = g(x, y, z_{i-1}) \text{ for } i = 1 \dots l, \quad (6)$$

for  $l$  a non negative integer and the function  $g$  defined as:

$$g : \mathbf{R}^3 \rightarrow \mathbf{R} \\ (x, y, z) \mapsto (y^2 + z^2)/x$$

$z_0 = z_0(x, y)$  is meant  $z_0$  is initialised for some values of  $x$  and  $y$ . For given values  $x = 3, y = 2$  and an initial guess  $z = 0.5$ , the implicit equation

$$z = g(x, y, z)$$

has a solution  $z_* = z_*(x, y) = 1$  and  $\nabla g(x, y, z_*) = (-1, 1)$ . When the code in equation 6 is mechanically differentiated using for example TAPENADE 2.1, we observed:

- if the initial guess is within a radius of the solution that leads to convergence, then the AD generated iteration converged to the correct derivative.
- if the initial guess is closer to the solution, say the initial value of  $z = 1$ , then the derivative iteration converges in one iteration to  $\nabla g(x, y, z_*) = (-1/3, 1/3)$ , which is wrong.

This means the assumption made by most AD tools to use the same number of iterations taken by the original iterative process for the derivative one is fair but may lead to wrong derivatives in certain cases. As suggested in [4], the AD tool ought to augment the convergence criterion to account for derivative convergence.

In summary, validating derivative calculation via AD can be difficult in the presence of non-differentiable functions and iterative solvers. It is hoped future AD tools will help spotting such anomalies and raising warnings to the AD user since, to our knowledge, there is no automatic ways of solving these issues.

## 1.2 About Efficiency

In theory, the fundamental idea of AD is simple: consider a computer code as a composition of elementary functions and differentiate it using the chain rule. The main advantage of this technique is that it provides algorithms that exhibit better accuracy and performance compared to finite differencing and that it is more reliable than hand-coded derivatives, which are error prone. In practice, there is a scope for AD tools to improve further in producing fast and reliable derivatives. Most of these improvements concern code optimisation. New algorithms have been developed and implemented into some AD software. These include the followings:

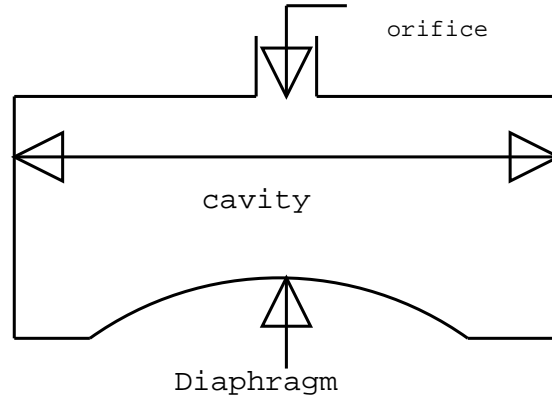
- Dependency analyses which determine the set of active variables and procedures. This eliminates many redundant calculations e.g., creating derivative objects that are a priori zero or adding/multiplying zero, see for instance [2, 6, 16].
- In-out analyses [16] which determine sets of active variables or required variables for the reverse sweep of the adjoint mode at subroutine level.
- TBR (To-Be-Recorded/Recomputed) analyses [6, 15] that enable reduction of the tape size [computing load] for the reverse mode AD. These analyses determine which values of variables are needed by the reverse sweep of the adjoint mode and hence need be stored or recomputed.
- Automatic checkpointing performed at subroutine call levels. Although not optimal, this strategy allows for a fair trade-off between storage and recomputation [16].
- Providing 'directives' facilities for the user to exploit certain insights of the code to be differentiated (e.g., parallel loops) [6].

However, hand-tuning the AD generated code can give further performance as illustrated in Section 3.1 of this report. In the next section, we focus on applying AD into the DGRINS code that implements a time-dependent CFD solver modelling a two-dimensional synthetic jet actuator.

## 2 DGRINS: A Time-Dependent CFD Solver

In previous work [21], the unsteady Navier-Stokes were discretised using a cell-centered finite volume approach and solved on a moving mesh. Figure 2 shows the chamber containing the air flow. The diaphragm is forced to move in a sinusoidal oscillation with given period  $p$  and amplitude (as a percentage of its length)  $a$ . Boundary and internal mesh points  $\mathbf{x}_i$  are moved into position  $\mathbf{x}_i^{new} = \mathbf{x}_i + \Delta \mathbf{x}_i$ , where  $\Delta \mathbf{x}_i$  is prescribed for boundary mesh points and for internal mesh points  $\Delta \mathbf{x}_i$  is given by,

$$\Delta \mathbf{x}_i = \frac{1}{D_i} \sum_{j \in \text{Nbr}(i)} \frac{1}{|\mathbf{x}_i - \mathbf{x}_j|} \Delta \mathbf{x}_j, \quad \text{with } D_i = \sum_{j \in \text{Nbr}(i)} \frac{1}{|\mathbf{x}_i - \mathbf{x}_j|}, \quad (7)$$



**Fig. 2** Schematic of a synthetic jet flow

where  $\text{Nbr}(i)$  represents the neighbouring points of  $i$ . This “smoothing” is repeated 50 times to ensure converged mesh movement.

In the numerical simulation, the residual  $\mathbf{R}_i(\mathbf{q}, \mathbf{x})$  for cell  $i$  is obtained using a finite-volume semi-discretisation of the Navier-Stokes equations in the form of  $\partial V_i \mathbf{q}_i / \partial t = \mathbf{R}_i(\mathbf{q}, \mathbf{x})$  where  $V_i$  and  $q_i$  are respectively the volume and conserved variables at cell  $i$ . By using a backward Euler scheme and introducing a pseudo-time step  $\tau$ , we obtain the following nonlinear system for  $\mathbf{q}^{n+1}$

$$V_i^{n+1} \frac{\partial \mathbf{q}_i^{n+1}}{\partial \tau} = \mathbf{R}_i(\mathbf{q}_i^{n+1}, \mathbf{x}^n) - \frac{V_i^{n+1} \mathbf{q}_i^{n+1} - V_i^n \mathbf{q}_i^n}{\Delta t}. \quad (8)$$

This system is solved by matching to steady state in  $\tau$  using low-storage 4-stage Runge-Kutta scheme. The objective function is defined as the time averaged specific kinetic energy  $F = \frac{1}{T_{max}} \int_{t=0}^{t=T_{max}} (u_k^2 + v_k^2) dt$  where  $k$  is the index of a nominated cell in the freestream above the cavity,  $(u_k, v_k)$  is the cell’s velocity vector and  $T_{max}$  is taken as 5 periods of the diaphragm motion. The numerical code called DGRINS can be sketched as follows:

```

Read in mesh geometry x (nodes, cells, faces)
Initialise flow variables and boundary conditions
Read in design variables designvars : a, p
Read in number of time-steps N and k
Set F = 0
For t from 0 to N
  Move the mesh boundary using a sin scheme
  Update interior mesh then cell and face information
  While (not converged) ! FIXPOINT
    Converge the flow variables using a RK4 solve:
      Compute residual R = r(q_j) for each cell j
      Update the flow variables q_j using R
  End While
  Update F = F + u_k^2 + v_k^2
EndDo

```

**Fig. 3** A Sketch of the time-dependent CFD code DGRINS

To maximise the kinetic energy represented by the objective function  $F$ , we need the gradient  $\nabla F$  and therefore the differentiation of the DGRINS code representing the function  $F$ . The actual DGRINS computer code is about 4,500 lines of code and uses Fortran features such as dynamic allocation, modules, derived types and

array operations. These features make the differentiation challenging since they are not well handled by current AD softwares.

### 3 Differentiation of DGRINS

To differentiate the DGRINS code, we use the TAPENADE 2.1 AD software. TAPENADE 2.1 is a source-to-source translation tool handling a subset of Fortran 95. It provides single directional derivative computation using the forward or adjoint modes as well as Jacobian evaluation using the so-called *vector mode* that concurrently calculates multiple directional derivatives using the forward mode.

Note that TAPENADE 2.1 only provides a limited handling of Fortran 95 derived types, modules and intrinsic functions (e.g., `matmul` and `dot_product` are not currently handled within TAPENADE 2.1) but does not handle at all dynamic memory allocation. These language coverage limitations imply we need “clean up” the code before being differentiated. By code clean up, we mean rewriting the input code into an equivalent form that can be well handled by TAPENADE 2.1. We manually replace the dynamic allocation to static allocation by declaring large enough arrays to store data. We then transformed Fortran modules into common blocks. Finally, as carried out in [20], we transformed all Fortran derived types into a set of arrays. For example the following declaration of the derived type called `bound_type`:

```
type node_type
  sequence
  double precision :: X
  double precision :: Y
  integer :: env
  integer :: actnum
  integer :: cellnei(neib_num)
end type node_type
type(node_type) :: node(nodenum0)
```

was replaced by the set of declarations:

```
double precision :: node_X(nodenum0)
double precision :: node_Y(nodenum0)
integer :: node_env(nodenum0)
integer :: node_actnum(nodenum0)
integer :: node_cellnei(neib_num,nodenum0)
```

and all occurrences of expressions of the form e.g., `node(i)%X` or `node(i)%cellnei(j)` in the code’s statements are replaced by respectively `node_X(i)` or `node_cellnei(j,i)`. This ‘cleaned up’ is carried out using the SED [5] scripting language to rewrite the entire DGRINS code.

Moreover, the vector mode of the current version of TAPENADE 2.1 works well when we specify the shapes of arrays in array operations meaning that if `a` and `b` are one-dimensional arrays, we write the statement `a(:)=b(:)` rather than its equivalent form `a=b`. Note that each ‘clean up’ need be validated by ensuring the semantic of the input code is preserved. This is carried out by gradually testing the transformed code against its original version.

#### 3.1 Results and Discussion

The rewritten code has been differentiated by TAPENADE 2.1 using the adjoint mode and the vector mode to calculate the gradient of the objective function  $F : \mathbf{R}^2 \mapsto \mathbf{R}$ . The obtained codes were compiled with maximum compilation optimisations and run for a small mesh size of 654 nodes and 582 cells on a Sun Blade 1000 machine. Both AD generated codes gave consistent gradient values in line with the result from finite-differencing. However, the adjoint code (TAPENADE 2.1(adj) in Table 1) required 1.5GB of tape size and 2 hours 9 minutes to run. This gives a ratio of 255 between the run-times of the gradient and function evaluations.

Because, this ratio is too high, we ought to seek further ways of improving the adjoint code. A profile of the TAPENADE 2.1(adj)-generated code showed that about 90% of the CPU time is spent storing or retrieving data

from the tape. We then identified kernel routines with a higher percentage of the CPU time and operated the following changes from the input code:

1. **Removal of overwritten variables:** An overwritten variable may be unnecessarily stored into the tape by the forward sweep of the adjoint mode for codes with complex control flow since the static TBR analyses are *conservative*, meaning they provide approximate solutions in lieu of exact solutions.
2. **Making arguments of subroutines explicit:** The flux and boundary routines operate on sets of flow variables (pressure, density, temperature and velocities) for each mesh cell that are stored in two-dimensional arrays. Arguments of these subroutines were cell indices which we have changed to flow variable arrays. The advantage is that the TAPENADE 2.1 taping mechanism will push 5 double precision numbers into the stack instead of  $5 \times k$ , where  $k$  is the number of mesh cells.
3. **Making local copies of global data:** The input code uses large arrays storing mesh or flow information in common blocks. Flux routines operate on two sections of these large arrays at a time. Making local copies of these sections increase data locality and minimises memory traffic by reducing cache or TLB misses. This is good for the input code and of great help for the AD software.

After these manual changes, the runtime of the resulting input code (indicated as “new” in Table 1) was decreased by about 40%. Then, we use the new input code for TAPENADE 2.1 to regenerate codes calculating the gradient of the objective function  $F$ .

The obtained codes were compiled with maximum compilation optimisations and all performance results for a small mesh size 654 nodes and 582 cells are shown in Table 1.

**Table 1** Performance of the AD-generated Code, the timings are in (s)

Method	CPU( $\nabla F$ ) (s)	Tape_size	$\frac{\text{CPU}(\nabla F)}{\text{CPU}(F)}$	Difference
TAPENADE 2.1(adj)	7774.6	1.585 GB	254.9	$\mathcal{O}(10^{-11})$
One-sided FD (new)	57.6	—	3.1	$\mathcal{O}(10^{-6})$
TAPENADE 2.1(new, fwd)	75.2	—	4.0	0.0
TAPENADE 2.1(new, adj)	450.8	0.12GB	24.2	$\mathcal{O}(10^{-11})$
TAPENADE 2.1(new, adj, FP)	384.2	0.09GB	20.7	$\mathcal{O}(10^{-6})$

Considering the TAPENADE 2.1(fwd) results as being correct, Table 1 shows the difference between AD and FD results is about  $10^{-5}$  and FD has used 3 function evaluations to calculate the gradient. TAPENADE 2.1(new, adj) is 10 times faster than TAPENADE 2.1(adj) showing that simple changes to the input code can result in a huge performance improvement of the AD-generated code.

This percentage dropped to 36% with TAPENADE 2.1(new, adj). However, TAPENADE 2.1(new, adj) is 5 times slower than TAPENADE 2.1(new, fwd). We believe this is due to the overheads incurred by the taping required by the adjoint mode AD. Note that TAPENADE 2.1 adjoint uses a recursive checkpointing strategy performed at the subroutine level to trade-off off between storage with recomputation of intermediate values. The last line of Table 1 shows the improvement in performance obtained by hand-modifying TAPENADE 2.1 adjoint code to store the converged nonlinear system and adjoining it [4]. Here, we have used the same number of iterations in the reverse pass as for the forward pass. The resulting derivative values are as precise as those of FD and the discrepancies with other AD results is, we believe, due to the finite convergence tolerance used in the nonlinear iteration at each backward Euler step. The runtime and memory requirement performance of the mechanical adjoining used by default in TAPENADE 2.1 are improved respectively by a percentage of 15% and 25%.

In AD theory [12], gradients are cheaply calculated using the adjoint mode when the number of independent variables is fairly large. Here, we have two independents and the computation of the adjoint is dominated by the taping routines. The adjoint approach will be competitive when we have a large number of independent e.g., geometric design of a complex actuator.

## 4 Application to an Optimisation Test Case

We consider a case study aiming at maximising the kinetic energy of the upwards motion in a given cell of the mesh above the chamber by controlling the period  $p$  and the amplitude  $a$  of the oscillation. We constrained this optimisation problem by setting a maximum work that the system cannot exceed. This work  $C(p, a) = \frac{1}{T_{max}} \int_0^{T_{max}} \int_{\Gamma=diaphragm} P(t)n(t) \cdot \frac{dx}{dt} d\Gamma dt$  is obtained by integrating the surface force over the time steps. Our optimisation problem is,

$$\begin{aligned} & \max F(p, a) && \text{subject to} \\ & \begin{cases} C(p, a) \leq 1 \\ 10^{-3} \leq p \leq 10^{-1} \\ 10^{-2} \leq a \leq 10^{-1} \end{cases} \end{aligned} \quad (9)$$

We used MATLAB's `fmincon` function to solve the above optimisation. The results are shown in Table 2. At the initial guess, we have  $F(p, a) = 2.65D - 4$  and  $C(p, a) = 8.73D - 3$ . We observed `fmincon` using the gradients

**Table 2** Optima calculated by MATLAB's `fmincon`

Method	Initial Guess		Optimum		Opt. Value F(p,a)	Iter.
	p	a	p	a		
<code>fmincon</code>	1.D-2	1.D-2	0.0024	0.0229	0.0389	13
<code>fmincon(AD)</code>	1.D-2	1.D-2	0.01	0.1	1.4535	11

supplied by AD converged to a better solution in 11 iterations than that obtained using FD in 13 iterations. By taking the solution obtained by `fmincon` using FD as an initial guess, the optimiser converged in 5 iterations to the optimal solution obtained by `fmincon(AD)`. We believe fast and accurate first derivatives provided by AD in the optimisation has increased robustness of the `fmincon` routine.

## 5 Conclusion and Future Work

We have presented preliminary results in using AD to produce an adjoint solver for a time-dependent CFD code. This shows that AD is a useful tool as it facilitates the automatic generation of adjoint code and can enhance robustness of an optimisation algorithm that requires derivatives. Furthermore, we have shown that simple changes of the input code prior to differentiation can greatly enhance the efficiency of the AD-generated code. The "better" the function is coded, the faster the AD calculated derivative will be.

Future work include optimising the geometry of the actuator by varying the width or length of the orifice or the volume of the chamber. This implies computing the mesh sensitivities. Therefore we need to differentiate the mesh generator. By choosing a certain number of control points that can be moved around for a superior geometry, there is a scope to increase the number of independent variables and therefore the AD adjoint mode will be of great benefit. We will also investigate checkpointing schemes that can further enhance the performance of the adjoint code and then run the resulting adjoint solver using finer meshes.

**Acknowledgements** This work is supported by EPSRC under grant GR/R85358/01.

## References

- [1] C. Bischof, B. Lang, and A. Vehreschild. Automatic differentiation for MATLAB programs. *Proc. Appl. Math. Mech.*, 2(1):50–53, 2003.
- [2] C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
- [3] A. Carle and M. Fagan. ADIFOR 3.0 overview. Technical Report CAAM-TR-00-02, Rice University, 2000.
- [4] B. Christianson. Reverse accumulation and attractive fixed points. *Optimization Methods and Software*, 3:311–326, 1994.
- [5] D. Dougherty. *Sed & awk*. O'Reilly & Associates, Inc., 1997.

- [6] FastOpt. *Transformation of Algorithms in Fortran, Manual, Draft Version, TAF Version 1.6*, Nov. 2003. See <http://www.FastOpt.com/taf>.
- [7] H. Fischer. Special problems in automatic differentiation. In A. Griewank and G. F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 43–50. SIAM, Philadelphia, PA, 1991.
- [8] S. A. Forth and M. M. Edvall. *User Guide for MAD - MATLAB Automatic Differentiation Toolbox TOMLAB/MAD, Version 1.1 The Forward Mode*. TOMLAB Optimisation Inc., 855 Beech St 12, San Diego, CA 92101, USA, Jan 2004. See <http://tomlab.biz/products/mad>.
- [9] S. A. Forth, M. Tadjouddine, J. D. Pryce, and J. K. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Transactions on Mathematical Software*, 30(3):266–299, Sept. 2004.
- [10] Q. Gallas, G. Wang, M. Papila, M. Sheplak, and L. Cattafesta. Optimization of synthetic jet actuators. *AIAA paper*, (0635), 2003.
- [11] J. C. Gilbert. Automatic differentiation and iterative processes. *Optimization Methods and Software*, 1:13–21, 1992.
- [12] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA, 2000.
- [13] A. Griewank, C. Bischof, G. Corliss, A. Carle, and K. Williamson. Derivative convergence for iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.
- [14] A. Griewank, D. Juedes, and J. Utke. Algorithm 755: ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw.*, 22(2):131–167, 1996.
- [15] L. Hascoet, U. Naumann, and V. Pascual. “to be recorded” analysis in reverse-mode automatic differentiation. *Future Generation Computer Systems*, 21(8), 2004.
- [16] L. Hascoët and V. Pascual. *The TAPENADE 2.1 user’s guide*. Inria Sophia Antipolis, Tropics Project, 2004, Route des Lucioles, 09902 Sophia Antipolis, France, Jan 2005. <http://www-sop.inria.fr/tropics/>.
- [17] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso. The complex-step derivative approximation. *ACM Trans. Math. Softw.*, 29(3):245–262, 2003.
- [18] G. D. Pusch, C. Bischof, and A. Carle. On automatic differentiation of codes with COMPLEX arithmetic with respect to real variables. Technical Memorandum ANL/MCS-TM-188, Argonne National Laboratory, Mathematics and Computer Science Division, 9700 South Cass Avenue, Argonne, IL 60439, June 1995.
- [19] L. B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.
- [20] M. Tadjouddine, S. A. Forth, and A. J. Keane. Adjoint differentiation of a structural dynamics solver. In M. Bücker, G. Corliss, P. Hovland, U. Naumann, and B. Norris, editors, *AD2004: Proceedings of the 4th International Conference on Automatic Differentiation*, Lecture Notes in Computational Science and Engineering, page To appear. Springer, 2005.
- [21] H. Xia and N. Qin. Dynamic grid and unsteady boundary conditions for synthetic jets flow. *43rd AIAA Aerospace Sciences Meeting and Exhibit, Reno, Nevada*, (AIAA 2005-106), 2005.