V European Conference on Computational Fluid Dynamics
ECCOMAS CFD 2010
J. C. F. Pereira and A. Sequeira (Eds)
Lisbon, Portugal,14-17 June 2010

ADJOINT CFD CODES THROUGH AUTOMATIC DIFFERENTIATION

D. Jones*, F. Christakopoulos and J.-D. Muller

School of Engineering and Materials Science, Queen Mary, University of London,
London, E1 4NS
*e-mail: dominic.jones@qmul.ac.uk

Key words: CFD, Discrete Adjoint, Automatic Differentiation

Abstract. *This paper presents some insights into constructing sensitivity algorithms, the tools required, what the constraints are and how to assemble the program. In this work Tapenade [2], a source transformation automatic differentiation tool, is used to obtain the sensitivity of a CFD system, whereby the independent and dependent variables are user-defined. Points on how to prepare the source code are detailed, with the aim of enabling the programmer to submit safe transformable code to Tapenade whilst retaining the use of modern Fortran structures, making use of modules, derived data types, pointers, etc.*

1 INTRODUCTION

By applying the chain rule consecutively to a differentiable system of equations, the sensitivity (Jacobian) of the system is obtained of all the dependent variables with respect to all the independent variables. Using the chain rule in its ordinary form, one obtains the Jacobian in a column by column manner. This is known as tangent mode. A tangent of the Jacobian describes the sensitivity of one independent variable to all dependent variables. The cost of constructing the Jacobian then is proportional to the number of columns in the Jacobian. However, for most optimization problems there are many independent variables and only a few dependent variables (thus many columns and few rows).

A preferred evaluation then of the Jacobian is row by row, which is achieved by taking the transpose of the complete chain of partial derivatives. This is known as adjoint mode, whereby each row represents the gradient of all independent variables with respect to one dependent variable. Tapenade performs both these tasks on user-provided code.

In certain types of CFD formulations, the solution of a sparse linear system is required [1]. Often the algorithm which performs this operation is called from a library, such as Sparskit [4], since it performs a general operation. It is undesirable to differentiate this operation, due to its complexity and the likelihood that the source code is not available in the first place. However, the task must be open to consideration. A highly optimized discrete adjoint sparse linear system solver would be a valuable contribution to mathematical libraries.

In this paper, a discussion is presented on the construction of the sensitivity of a non-linear system. Present capabilities of Tapenade are highlighted along with insight into how to use it effectively. Considerations on program structure in Fortran which is to be parsed or transformed by Tapenade are detailed, since Tapenade is not capable of transforming all programming structures available in the language.

2 CODE PREPARATION

2.1 Language Support

A given automatic differentiation (AD) tool does not necessarily recognize the entire syntax of the programming language it operates on. AD is fundamentally concerned with transforming mathematical statements. Bridging the discrepancy between the language features used in the code to be differentiated and the what the tool can parse/transform is perhaps the most significant obstacle to overcome.

If the code is being written from scratch, the following is good practice. Procedures to be differentiated should be pure, implying that all parameters are passed explicitly. Allocation within routines to be differentiated should be avoided. Certain control sequences, if present, will need modifying: the *forall* and the *do while* constructs both need to become *do* loops.

Best practice is to submit short example codes to the AD tool to see exactly what the tool can deal with, only being sure that a programming feature can be used once the resulting differentiated code is compiled and successfully tested (testing both forward (tangent) and reverse (adjoint) mode).

2.2 Modular Structure

Encapsulation of all subroutines within modules offers a number of benefits for both the original code and the differentiated code. All calls are type-checked, pointers may be passed, optional arguments and explicit argument naming may be used, array sizes can be implied. One module per file is a sensible arrangement, as this keeps one-to-one file/module correspondence for the differentiated code with the primal code. Include statements, whereby its contents contains derived type definitions, used outside modules ought to be avoided. These are best contained within modules. Finally, a file containing one module should have the name of the module as its file name. This helps with writing the makefile rules.

With the use of modules, compilation of files cannot be performed in any order; compilation must begin with the base module which has no dependencies, and on to the top level.

2.3 Derived Data Types

As these are very useful for collecting related data and can be the cause of many problems in AD, they are especially examined. Tapenade correctly parses derived data type definitions, such as

```
type::face_t
  integer::id=0
  real,dimension(:),pointer::x,norm
  real::area=0
}
```

Assuming this type is used in a differentiated routine whereby all floating point variables are active, Tapenade will not create a new type if `id` was not present. However, it is better to cause the tool to create a new type for ease of code readability. This is easily done by adding an integer into the type. The resulting differential type becomes

```
type::face_t_d
  real,dimension(:),pointer::x,norm
  real::area=0
}
```

If data is passed explicitly, declaration of differential variables must be handled by the user at the top level. In addition, it is useful to force the tool to create differential type definitions which it does not ordinarily see. Suppose a procedure calculates the convective flux, $\rho S\vec{v}\cdot\vec{n}$, as

```
conv_flux(dens,area,vel,norm)
  flux = dens*area &
    *dot_product(vel,norm)
}
```

and the routine is called by `conv_flux(dens,face%area,vel,face%norm)`. Upon differentiating the procedure, the AD tool does not produce a differential face type because it sees no relation between the data structure and the procedure called. However, the programmer probably wants such a structure to exist. In order to do this, a dependency routine is written and differentiated, along with the original procedure.

```
active_face_elements(face,x)
  type(face_t)::face
  real::x
  face%area = x
  face%norm = x
  face%x = x
}
```

When passing an array of derived types to a procedure, it is necessary to pass its size as well in order for Tapenade to correctly parse the code; i.e.

```
boundary(pde,ipde,i,ib)
  type(pde_t),dimension(:)::pde
  pde(ipde)%phi(ib)=pde(ipde)%phi(i)
}
```

must become

```
boundary(n_pde,pde,ipde,i,ib)
  type(pde_t),dimension(n_pde)::pde
  pde(ipde)%phi(ib)=pde(ipde)%phi(i)
}
```

Using pointers to point to elements of derived data types purely for the purposes of having a shorter token is commonly done. This is permitted by the Tapenade (it is parsed), but at present, in reverse mode the pointers cause a problem which can be fixed by a script afterwards. If the original code is of the form

```
do i=1,n
  phi => pde(ipde)%phi(i)
  phi = ...
}
```

either the code needs to be rewritten, becoming

```
do i=1,n
  pde(ipde)%phi(i) = ...
}
```

or the pointer must be set pointing to a true address in the adjoint code at the earliest point, as in

```
phi => pde(ipde)%phi(1) !! added
...
do i=1,n
  call PUSHPOINTER(phi)
  phi => pde(ipde)%phi(i)
  ...
}
```

2.4 Iterators

For computing surface and volume integrals loops are performed over all elements. The order in which this is done is irrelevant, so can be treated as iteratively independent. However, whilst the programmer may know this, Tapenade is not yet able to determine this, so a pragma is required, of the form

```
!$AD II-LOOP
do i=1,n
  ap(i)=ap(i)+aw(i)+ae(i)
}
```

This pragma becomes useful when reverse mode is used.

2.5 Useful Tools

In addition to the AD tool, Make and the C preprocessor significantly ease the building of the program. In the case of Fortran, the C preprocessor built in to the compiler ought to be used, if a preprocessor is required. If modifications to the differential code is required, Perl offers much regular expression support, though often Sed is sufficient.

3 ASSEMBLY

3.1 Calling Tapenade

In order to obtain the correctly differentiated code, Tapenade must be called properly, whereby the command is of the form

```
$tapenade -[bd] -head "subr1(vars)\(outvars) ..."
```

The code which contains the routines to be differentiated and all its dependencies must be passed to Tapenade when the program is invoked.

The key issue is what the `vars` and the `outvars` are to be set as. To create the differential of all dependent variables with respect to all independent variables, these are left blank. This is the simplest choice, though the resultant routines may be calculating many derivatives which are not needed.

As a first premise, `vars` should be the list of variables from which derivatives are wanted with respect to and `outvars` should be the list of variables about which derivatives are wanted. However, one needs to look at the context in which the derivate code is to be used to be sure about how to invoke the AD tool properly.

Consider the following example:

```
subr(x,f)
  f = f + x
}
```

To differentiate this, one may simply invoke

```
tapenade -[bd] -head "subr(x,f)\(f)" <file>
```

since this reflects the intent of the parameters, producing

```
subr_d(x,xd,f,fd)
  fd = fd + xd
  f = f + x
}
```

for tangent mode and

```
subr_b(x,xb,f,fb)
  xb = fb
}
```

for adjoint mode. Suppose the subroutine was called within a loop; for the adjoint code, `xb` would overwrite its state prior to entry. This may not be the desired behaviour as other procedures may be contributing to this term at an earlier stage. If this is the case, Tapenade should instead be invoked as

```
tapenade -b -head "subr(x,f)\(x)" <file>
```

resulting in

```
subr_d(x,xb,f,fb)
  xb = xb + fb
}
```

3.2 Using the Transformed Source Code

After Tapenade is called, it outputs the primal and derivative source code. As a result, two versions of the primal now exist and will cause a conflict at the linking stage. In the case where only tangent or adjoint code is required, the program may be compiled with the output code from Tapenade. If both tangent and adjoint code are required for the same program then it is best to transfer the tangent procedures into the adjoint source code files, ensuring AD suffices are the same for both modes.

The output code then is then compiled and linked with the higher level routines. In the top level caller routines, it is helpful to use macros for compiling either the original code or the sensitivity code.

3.3 Non-linear System

It has been assumed so far that one already knows what to differentiate in the system. However, there may be some uncertainty in this or an alternative approach to differentiating the system may be desired. The latter reason is the purpose of the following discussion. Here, the desire is to avoid having to differentiate the linear solver in addition to the other dependent subroutines to obtain the sensitivity of a non-linear problem. This case arises particularly in incompressible CFD algorithms. Treatment of this problem is well documented for compressible solvers, whereby the residual of the non-linear system is computed. A clear discussion of how to obtain the adjoint is found in [3].

To obtain the sensitivity of a CFD system, at some point, the differential of the discretized momentum equation will need solving

$$\frac{\partial}{\partial \vec{x}}\left[A(\vec{u},\,\vec{x})\;\vec{u}(\vec{x})\right] = \frac{\partial}{\partial \vec{x}}\left[\vec{b}(\vec{u},\,\vec{x})\right] \tag{1}$$

becoming (dropping the dependencies)

$$A\frac{\partial \vec{u}}{\partial \vec{x}} = \frac{\partial \vec{b}}{\partial \vec{x}} - \frac{\partial A}{\partial \vec{x}}\vec{u} \tag{2}$$

This system can readily be constructed in pure tangent mode without having to differentiate the linear solver. The attainment of each differential term can be accounted for: the first term on the right-hand side coming from the differential of source term construction in tangent mode, and the second term on the right-hand side coming from the differential of matrix construction in tangent mode.

However, the principle interest is whether or not the transpose of this system can be performed in purely adjoint mode without differentiating the solver. Placing the problem in context, the differential of the entire system is of the form

$$\frac{\mathrm{d}J}{\mathrm{d}\vec{x}} = \frac{\partial J}{\partial \vec{u}}\frac{\partial \vec{u}}{\partial \vec{x}} + \dots \tag{3}$$

where J is the objective function, which at least depends on the velocity field. Substituting in Eq. (2) and transposing gives

$$\begin{aligned}
\frac{\mathrm{d}J^T}{\mathrm{d}\vec{x}} &= \frac{\partial \vec{u}^T}{\partial \vec{x}} \frac{\partial J^T}{\partial \vec{u}} + \dots \\
&= \left[\frac{\partial \vec{b}}{\partial \vec{x}}^T - \vec{u}^T \frac{\partial A^T}{\partial \vec{x}} \right] \left\{ A^{-T} \frac{\partial J^T}{\partial \vec{u}} \right\} + \dots \\
&= \left[\frac{\partial \vec{u}^T}{\partial \vec{x}} \frac{\partial \vec{b}}{\partial \vec{u}}^T + \frac{\partial \vec{b}}{\partial \vec{x}}^T - \vec{u}^T \left(\frac{\partial \vec{u}^T}{\partial \vec{x}} \frac{\partial A^T}{\partial \vec{u}} + \frac{\partial A^T}{\partial \vec{x}} \right) \right] \left\{ A^{-T} \frac{\partial J^T}{\partial \vec{u}} \right\} + \dots \qquad (4)
\end{aligned}$$

where the term in braces indicates the solution of a linear system, whose solution is referred to as the adjoint flow field, ψ, i.e.

$$A^T \psi = \frac{\partial J^T}{\partial \vec{u}} \qquad (5)$$

Obtaining the adjoint flow field is simple enough; the adjoint of the cost function is first obtained, then the result is passed to the linear solver, with the transposed matrix, returning ψ. The remaining terms however, require some analysis.

Within the brackets there are two main terms, relating to the right-hand side, b, and the matrix, A. From the fully expanded form of Eq. (4), both $\frac{\partial \vec{b}}{\partial \vec{x}}^T$ and $\frac{\partial A}{\partial \vec{x}}^T$ are readily computed from the tangent source and matrix construction procedures. The cost of computing them is proportional to the number of independent variables, though the cost is relatively cheap, especially for the matrix derivative.

Alternatively, the adjoint of both the source and matrix construction can be computed, obtaining all the partial derivatives with respect to b and A, but at potentially significant cost (proportional to the mesh size).

However, this still leaves the term $\frac{\partial \vec{u}}{\partial \vec{x}}^T$. Since A^{-T} is not computed, there is no obvious way of obtaining this term other than performing

$$\begin{aligned}
\frac{\partial \vec{u}^T}{\partial \vec{x}} &= \left[\frac{\partial \vec{b}}{\partial \vec{x}}^T - \vec{u}^T \frac{\partial A^T}{\partial \vec{x}} \right] \psi \, \frac{\partial J}{\partial \vec{u}}^{-T} \\
&= \left[\frac{\partial \vec{b}}{\partial \vec{x}}^T - \vec{u}^T \frac{\partial A^T}{\partial \vec{x}} \right] \psi \, \left\| \frac{\partial J}{\partial \vec{u}} \right\|^{-2} \frac{\partial J^T}{\partial \vec{u}} \qquad (6)
\end{aligned}$$

where $\frac{\partial J}{\partial \vec{u}}$ in practice is a real vector. This approach implies that an iterative algorithm is required to compute the result. Even if Eq. (6) offers a reasonable way of computing it relatively cheaply, the whole process has already become laboured enough, indicating that an adjoint linear solver would considerably simplify matters.

Assuming that an adjoint linear solver was available, obtaining the system sensitivity would be no more than (more or less) assembling the procedure calls in the correct order, which is the reverse of the normal order. Taking a functional approach, the original algorithm may look like

$$\begin{aligned} &A = f_A(x,\, u);\; b = f_b(x,\, u) \\ &u = f_u(A,\, b) \\ &J = f_J(u, \ldots) \end{aligned} \tag{7}$$

where the function f_u is the linear solver. Thus obtaining the adjoint of each function and adopting the notation of $\overline{\phi} \equiv \frac{\partial J}{\partial \phi}^T$, the pure adjoint may be written as

$$\begin{aligned} &f_J^R(u, \overline{u},\, J, \overline{J}, \ldots) \\ &f_u^R(A, \overline{A},\, b, \overline{b},\, u, \overline{u}) \\ &f_b^R(x, \overline{x},\, u, \overline{u},\, b, \overline{b});\; f_A^R(x, \overline{x},\, u, \overline{u},\, A, \overline{A}) \end{aligned} \tag{8}$$

where $\overline{J}$ is set to one, causing f_J^R to return $\overline{u}$. This becomes the input to f_u^R and the procedure continues until the two contributions to the sensitivity are obtained.

At present, an adjoint linear solver is not available to the authors, so the sensitivity example computed in adjoint mode is a linear problem (fixed velocity field). As the tangent mode does not require a differentiated linear solver, the sensitivity of the complete Navier-Stokes equations can be solved, and is presented below.

4 TANGENT MODE EXAMPLE

The case presented is taken from [1] though with a slightly modified inlet condition. It is solved with a cell centred finite-volume code, suitable for incompressible flow.

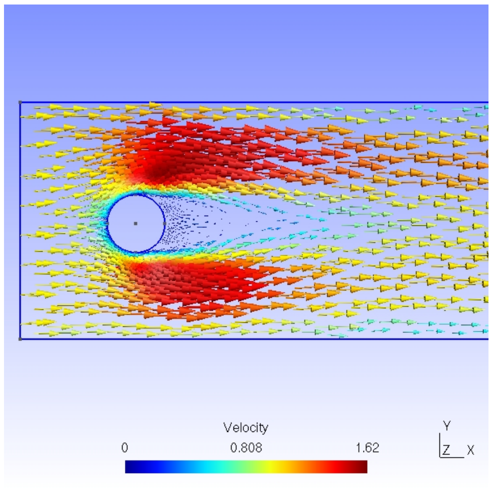
To validate the tangent formulation of the sensitivity algorithm, the simplest test is to find error between the Jacobian by finite differencing and the Jacobian by tangent mode. Progressively reducing the interval step size in the finite difference approximation, the error should drop to approximately 10^{-5} then increase again as the step becomes too small.

The adjoint formulation may be validated in the same manner, though it is better to validate it against an already validated tangent formulation. Here the error should be close to the order of the floating point precision.

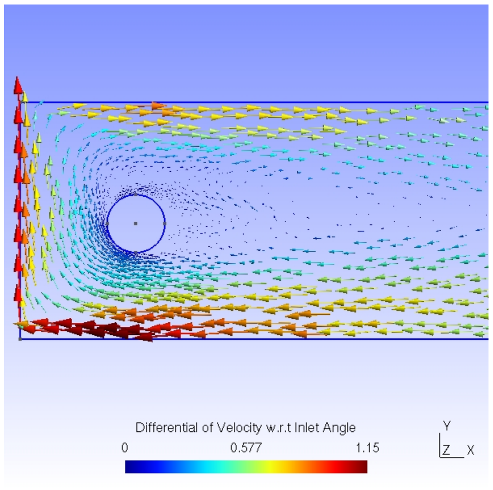
4.1 Flow around a Cylinder

The drag coefficient sensitivity with respect to the inlet velocity angle is determined using the tangent formulation of the Navier-Stokes equations for incompressible flow. The arrangement of the sensitivity algorithm is the same as that for the primal algorithm. Convective and diffusive terms are calculated to second order accuracy and the SIMPLE scheme for pressure-velocity coupling is implemented.

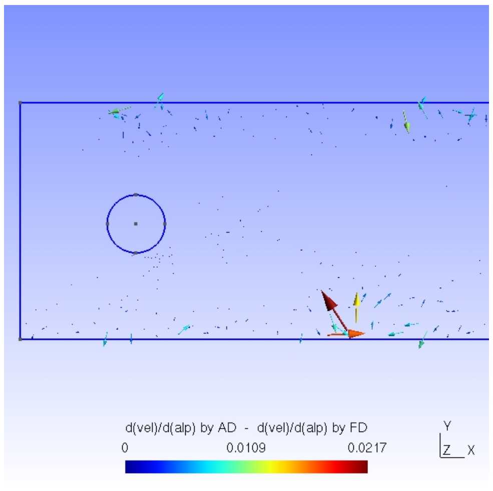
Figure (1) shows the fields of velocity, velocity sensitivity $\frac{\partial \vec{u}}{\partial \alpha}$ and sensitivity error $\frac{\partial \vec{u}}{\partial \alpha}_{\text{Tang}} - \frac{\partial \vec{u}}{\partial \alpha}_{\text{FD}}$ of steady fluid flow around a cylinder (Re=20). The sensitivity error shows small regions of relatively large error, though this is not surprising as the finite differencing is only first order accurate and the system is non-linear. In the majority of the domain the error is very small and the error norm is below 10^{-5}.



(a) Velocity field



(b) Velocity sensitivity



(c) Sensitivity error

Figure 1: Flow and its sensitivity around a cylinder

5 CONCLUSIONS

A brief overview of obtaining the sensitivity of a system using automatic differentiation has been presented. Practical aspects on preparing the source code have been stressed as this can be the most time consuming step in the process. Analysis of the derivative of non-linear systems has been given as this is the most complex part of a CFD code and it has been shown that to obtain the sensitivity by the adjoint mode practically, the adjoint of the linear system solver is required. This will most likely be the topic of future work.

6 ACKNOWLEDGEMENTS

This research is part of the European project FLOWHEAD (Fluid Optimisation Workflows for Highly Effective Automotive Development Processes), funded by the European Commission under THEME SST.2007-RTD-1. http://flowhead.sems.qmul.ac.uk/

REFERENCES

[1] J. H. Ferziger and M. Perić. *Computational Methods for Fluid Dynamics.* Springer, third edition, 2002.

[2] L. Hascoët and V. Pascual. Tapenade 2.1 user's guide. Technical Report 0300, INRIA, 2004.

[3] D. J. Mavriplis. A discrete adjoint-based approach for optimization problems on three-dimensional unstructured meshes. *AIAA*, 2006.

[4] Y. Saad. Sparskit: A basic tool kit for sparse matrix computations. *Version 2*, 1994.