CFD Development with Automatic Differentiation

Dominic Jones[*], Jens-Dominik Müller[†] and Sami Bayyuk[‡]
Queen Mary University of London, London, E1 4NS, UK
ESI North America, Huntsville, AL 35806, USA

This paper documents the approach used to develop the sensitivity algorithm of a commercial CFD code from ESI written in Fortran 90/95 using INRIA Tapenade version 3.6. A primitive CFD code was written in tandem with this work which is consistent with the basic algorithms used in the commercial code to present as a proof-of-concept the feasibility of an implementation and to assess the sensitivity data computed by the generated algorithms.

Introduction

Development of adjoint CFD solvers started with the *continuous* approach[1] where the N-S equations are differentiated, then transposed and equipped with boundary conditions, then discretised. This approach is currently the dominant approach when discretising the incompressible flow equations.[2,3] While this approach can be rapidly implemented, it often leads to problems with numerical stability. A more insidious issue is the fact that this re-discretisation implies different grid resolution requirements for the adjoint equations. For a turbulent case without wall functions[4] find grid convergence for adjoint gradients at $y^+ \leq 0.1$, compared to typical requirements of the primal at $y^+ \leq 1-2$.

More recently, the *discrete* approach has been favoured for compressible flow discretisations, where instead of a solution to a set of adjoint equations, we compute the direct differentiation of the discretised flow equations.[5,6] The process is straightforward and methodically applicable, so it can be automated. The resulting adjoint code also by construction inherits the linear stability of the primal CFD code. Finally, as the discrete adjoint is a literal differentiation of the primal code, it also inherits its grid requirements. The challenge in this approach however is that a 'black-box' application of a software tool to perform this automatic differentiation (AD) in reverse mode will typically result in very inefficient code with excessive memory and runtime requirements.

An alternative 'half-way' house approach has been advocated by a number of authors for the fully coupled time-stepping schemes of the standard finite volume compressible discretisations.[7–9] Here AD is applied to the flux computations only and the differentiated flux routines are assembled into hand-coded iterative loops. The fully coupled update schemes typically used for compressible flows can be derived from the primal time-stepping without too much difficulty. This is not however the case for the predictor/corrector time-discretisation with partial updates typical of current incompressible flow algorithms that makes extensive use of linear solvers inside a non-linear iteration. A first algorithm for an adjoint pressure-correction scheme that follows this modified time-stepping approach has been proposed recently.[10]

At present, no major CFD software vendor uses automatic differentiation by source transformation to construct the sensitivity algorithm of their N-S implementation, but rather implements a continuously differentiated approach[11] or hand-codes its discrete derivative (a speculation).[12] No AD tool is currently capable of handling such a piece of software in a black-box manner, nor should such capability be readily expected for a number of reasons, of which some are discussed in[13] and.[14]

Perhaps one reason for this is false expectations of what *automatic* differentiation tools are expected to do for the software developer such that a sensitivity algorithm can be successfully produced. If this is the case, then what ought the developer really expect the AD tool to do for him? A safe answer to this, from

[*]Research Associate, School of Engineering and Materials Science.
[†]Senior Lecturer, School of Engineering and Materials Science.
[‡]Senior Developer, ESI North America

the experience of the authors, is the differentiation of a self-contained algorithm written in a single language with no third-party library references, without any non-standard language features and preferably without dynamic memory management.

If the developer is to take such a safe approach then he must find a way of extracting from the original software its numerical algorithm, modifying it such that it is in a fit state for differentiating, differentiate using a suitable AD tool[15] then piece the software back together again to include its sensitivity algorithm.

Breaking apart and putting back together such pieces of software is no small task. Furthermore, the implementation of these tasks would be tailored to the software itself and the kind of differentiation being performed. However, despite this tailoring, there are generic tasks which AD tools ought to (but appear not to) do, such as provide an accessible trace of all active routines and variables from some specified top-level routine and provide an extensible pragma facility to modify the differentiation process.

How the code is to be differentiated strongly relates to how readily the complete program can be assembled and how readily it can be algorithmically optimized. At present, these two desirables appear to be partially mutual trade-offs. On the one hand, the approach of operator overloading has grown in popularity with AD tool developers,[16–18] even to the extent of incorporating it into the source code compiler itself,[19] but comes at the expense of the programmer having an opaque grasp on what exactly is happening in the algorithm (in particular the adjoint) thus losing the capability (at least partially) of optimisation, which as a result must instead be a task taken up to some extent by the AD tool developer via trace analysis[20,21] or convoluted programming techniques.[22]

On the other hand, source code transformation AD[23,24] offers a free reign to the developer over what code is made visible to the AD tool, how it is to be differentiated and what should be done with the resulting differentiated code, potentially enabling the construction of highly efficient algorithms.[25] It is this approach which is considered herein.

This paper presents the first attempts at developing a framework for preparing source code for differentiation by an AD tool and for editing the generated source code such that the sensitivity algorithm is optimized and the compiled code can slot in to the original program just like a library extension (figure 1). Much of the overall process is automated via standard programming utilities. Secondly, validation techniques are presented along with some insights into the behaviour of the adjoint computation.

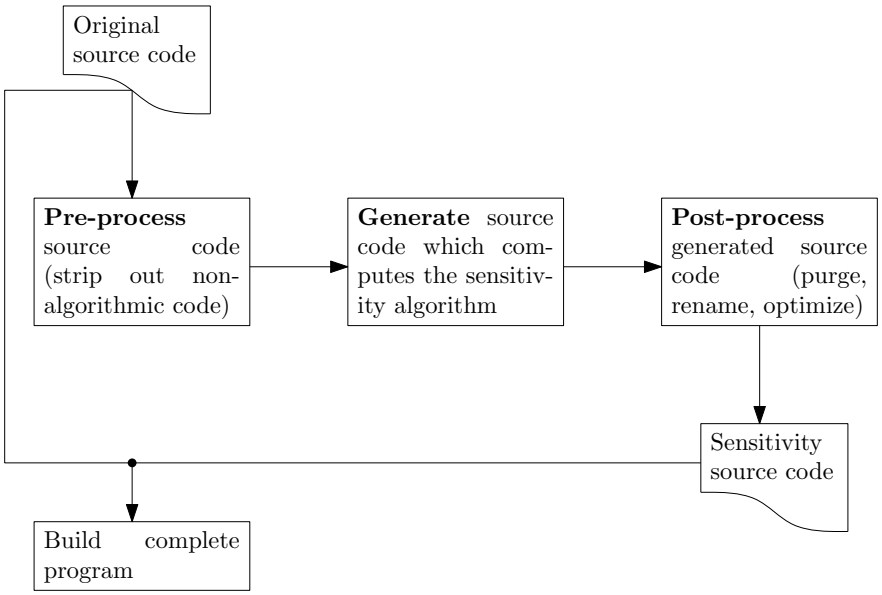


Figure 1. Construction of the complete program with sensitivity functionality

Identifying the algorithm

CFD software typically has the following broad divisions: mesh and field data input/output, mesh management, geometry evaluation, PDE construction and solution and post-processing facilities. Of these sections, only the geometry metrics computation and PDE construction and solvers are of interest. For the

N-S computation, this may look something like the sequence shown in figure 2.

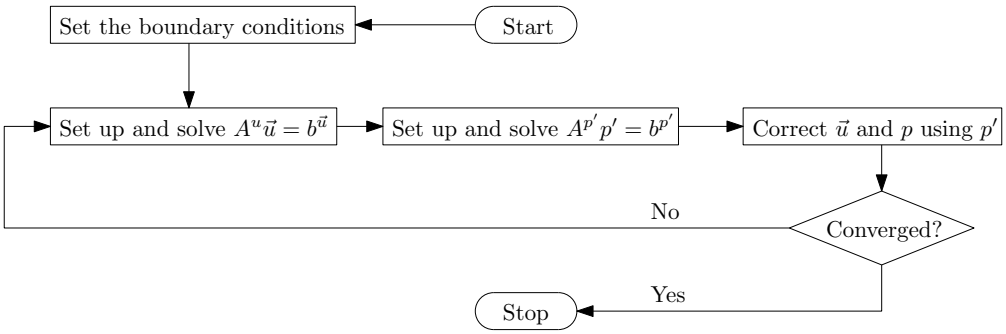


Figure 2. Minimal Navier-Stokes algorithm

It is this, either holistically or elementally, which must be differentiated. Before that can take place, the source code that contains that algorithm must be identified and then purged of all non-traceable references, such as MPI communication, linear system solver libraries and any other auxiliary library functions.

Tracing the top-level function

Before supplying the source code to an AD tool for differentiation, one must first know what source code the tool needs to *see* in order to fully trace the dependencies of the top-level function. In a simple program such tracing can be done manually, and if desired, dead code can be omitted from the outset. Scale up to a million line code and problems arise. Suppose in an initial attempt, one wishes to differentiate the N-S algorithm in its simplest state (call it the kernel, figure 2), i.e. turning off limiters, high-order interpolation, advanced physics models, etc, then one would prefer to only provide to the AD tool the code which contains the kernel. But where is this source code contained and what is done to calls in the kernel to advanced features? Unfortunately a solution to this problem is not provided by any AD tool known to the authors, but a reasonable method can still be constructed via the Tapenade AD tool.[24]

Tapenade dumps the call graph of submitted source code when the `-html` flag is used, whether or not the source code is compilable. For references to undefined tokens, such as a subroutine which is called but whose definition is not visible, they are listed as `external`. With this information the body of available source code can be scanned until definitions of missing tokens are found. The source code files which contain these can then be added to the list of files made visible to Tapenade. The only caveat to this method at present is that if a token is aliased, the alias name appears in the call graph, not the true name, i.e. in Fortran, if there is a line such as `use mod, only: alias => token`. A partially automated flow graph for tracing the call graph is presented in figure 3. The manual aspect this is the stripping out of code which is found in a source code file but is not part of the searched for definition.

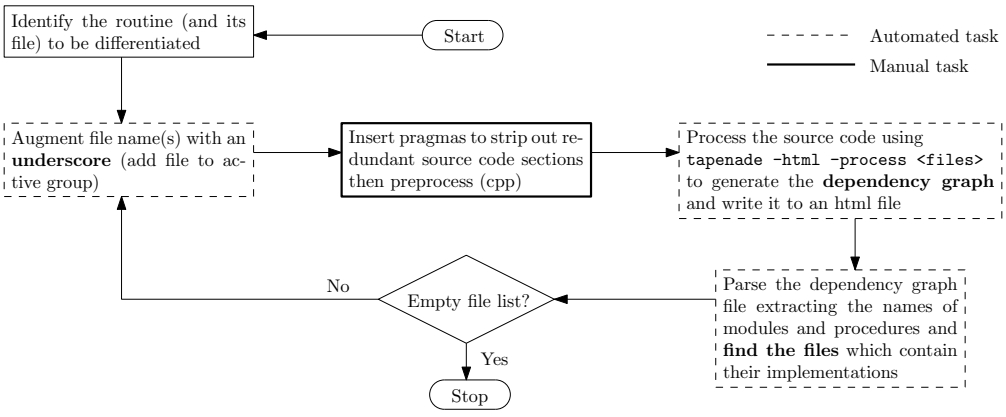


Figure 3. Finding active source code files

Pruning the source code

Filtering out what code is to be made visible to the AD tool is performed by tracing the top-level function to be differentiated. However, prior to including new files into the body of source code being introduced to the AD tool, relevant pragmas may need introducing. The task of the pramgas, which is shown in figure 4, is to further crop the code by removing passive definitions, replacing the body of active non-differentiable routines and removing references to extended algorithmic features in the active code which may not be desired in the sensitivity algorithm (at least initially).

```
module matrix_utils_m
contains

#ifndef STRIP_AD
!! passive
subroutine print_matrix(mat)
  ...
end subroutine
#endif

!! active non-differentiable
function coo_to_csr(i,j,ja,ia) result(kij)
#ifndef STRIP_AD
  ...
#else
  kij = 0 ! retain a trivial dependency
#endif
end function

!! active differentiable
subroutine matrix_setup(a_ij,phi,ja,ia,res)
  ...
#ifndef STRIP
  if(use_advanced_feature) call adv_feature()
#endif
end subroutine
end module
```

Figure 4. Removing redundant source code

Two different pragmas are deliberately used here. The first, `STRIP_AD`, is only defined when pre-processing the source code prior to submitting it to the AD tool, whereas the second, `STRIP`, is defined when both building the ordinary program and pre-processing for the AD tool. This distinction enables the ordinary program to be compiled and run in its reduced form without introducing linkage problems.

To clarify which non-critical features of an algorithm may be stripped out in order to simplify the initial attempt a differentiation, it is helpful to compare the complete call graph generated by Tapenade with a profiler output (such as `gprof`) after running the code for a simple flow case. Any routines which are not listed in the profiler output but are in the static call graph can be safely stripped out in the preparation of the primal code (figure 5).

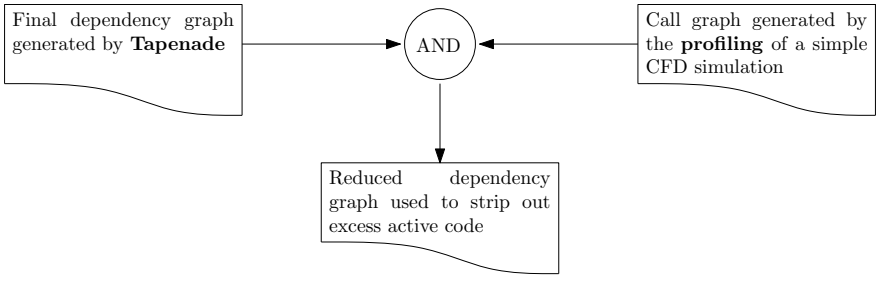


Figure 5. Clarifying which routines can safely be removed via profiling

Programming standards

A final preparation task to perform prior to differentiating the source code is to check that the source code to be submitted conforms to the programming standard in which it is written in, avoiding all redundant, obsolescent and deleted features[26] *and* that it conforms to the standard recognised by the AD tool. In the case of Fortran it is prudent to treat all intrinsic tokens as keywords, i.e. that overriding them is forbidden. This can be forced by using the `intrinsic` keyword.

In the present version of Tapenade (version 3.6) there are certain points to be aware of:

1. using `save` as an attribute of a module scope causes incomplete differentiation,
2. the `case default` must be the last case,
3. the differentiation of statements involving pointers is under development.

Post-processing generated source code

Once the differentiated code is produced, the assembly of it with the remaining original code is required. Before that step is done though, it is useful to edit the differentiated code for the following reasons:

1. inherit original modules into their generated derivative modules,
2. purge generated code of all definitions of primal equivalent routines and data (except private data),
3. ensure that all references to primal routines and data refer to original code and not to equivalent generated code,
4. identify and remove generated derivative type definitions and replace associated declarations using that type with the original type,
5. for the adjoint of linear system solvers, use the hand coded alternative[27] (this must be properly converged at each invocation),
6. in adjoint code, identify fixed-point iterations, reconfigure it to record once and restore once active primal variables,
7. in adjoint code, identify active quantities which can be assumed to behave like constants and remove their associated adjoint computation.

All but the last two steps are strictly necessary to perform if the general approach of this work is to be followed. The last two points should be implemented, especially regarding fixed-point iterations, otherwise memory space may easily be exhausted. An example of how the source code is transformed is presented in figure 6.

Original code

```
module pdes_m
use base_m

character(3), &
  private::fmt="csr"

type::pde_t
  real,dimension(:), &
    allocatable::phi,rhs
  real::reduc
  integer::max_iter
end type

type(pde_t)::pres, vel

contains

subroutine navier_stokes()
  ...
end subroutine
end module
```

Generated code (Tapenade)

```
module pdes_m_b
use base_m_b

character(3),private::fmt="csr"

type::pde_t
  real,dimension(:), &
    allocatable::phi,rhs
  real::reduc
  integer::max_iter
end type

type::pde_t_b
  real,dimension(:), &
    allocatable::phi,rhs
end type

type(pde_t)::pres, vel
type(pde_t_b)::pres_b, vel_b

contains

subroutine navier_stokes_c_b()
  ...
end subroutine

subroutine navier_stokes_b()
  call gradient_c_b()
  call setup_mat_rhs_c_b()
  call solve_c_b()

  call solve_b()
  call setup_b()
  call gradient_b()
end subroutine
end module
```

Modified (in-place) generated code

```
module pdes_m_b
use pdes_m ! import original
use base_m_b

character(3),private::fmt="csr"

type(pde_t)::pres_b, vel_b

contains

subroutine navier_stokes_b()
  call gradient()
  call setup_mat_rhs()

! A.x = b:
! not necessary since a
! fixed point is assumed
! to have been reached
!   call solve_c_b()


! adjoint of A.x = b:
! manual implementation
  call solve_rev()

  call setup_b()
  call gradient_b()
end subroutine
end module
```

Figure 6. Example of the post-processing performed on generated source code. N.B. The _c_b suffix refers to the generated interpretation of original code.

Assuming certain quantities behave like constants in the reverse sweep of the adjoint calculation becomes useful when their net contribution is small but their presence may destabilise the computation or is expensive. An example of this is a gradient limiter. The limiter makes a small contribution but is expensive to compute and for most limiters non-differentiable. In this case it is beneficial to ignore its variation in the adjoint.

Final Preparatory Remarks

Once the derivative code is generated, the next task is to incorporate it into the program as an executed function. For successful use of the derivative function all derivative dynamically allocated variables must be allocated and all derivative variables must be initialised (typically to zero, except for the independent variable, which is typically set to one). Determining what derivative variables require allocating can be a very difficult task for large complex codes and is at present determined by using debugging utilities.

To minimise the amount of differentiation being performed, the AD tool can be invoked with the function to differentiate *and* stating the independent and dependent variables of whose derivatives are required. However, some preparation work may be required in order to specify the independent and dependent variables. In the example of figure 7, the subroutine `calc` cannot be differentiated with respect to `x` only, unless code refactoring is performed.

```
module calc_m
integer, parameter :: bx=1, by=2, bf=3

type blk_t
  integer :: ctrl
  real, dimension(:), pointer :: lst
end type

type(blk_t), dimension(:), pointer :: blk

contains

subroutine calc()
  real, dimension(:), pointer :: x, y, f
  x => blk(bx)%lst
  y => blk(by)%lst
  f => blk(bf)%lst
  f = x**2 + y**2
end subroutine
end module
```

Figure 7. Implicit parameters preventing the specification of required derivative arguments.

Validation Techniques

Discrete sensitivity computation, as opposed to its continuous complement, can be validated with a high degree of confidence due to the fact that the exact numerical derivative of the original algorithm is computed. The procedure of validating the discrete adjoint involves three steps:

1. compute the objective at its initial and perturbed state to generate a finite difference (FD) approximation of the sensitivity. If there are many state variables, a pertinent section may be taken. A suitable step width for FD will need to be determined first. Convergence to machine precision is recommended in order to perform accurate comparisons in the next step.

2. Using the same initial state(s), construct and compute the forward mode (tangent) sensitivity. Compare these results to those from finite differencing. If the sensitivities are very similar then validate the adjoint sensitivity. If not, reduce the algorithm to its simplest operations and re-test. If this solves the problem, systematically introduce the advanced sections of the algorithm until the cause of disparity is found. Once found, examine and re-code the original source code section into simpler statements. If this does not solve the problem, generate a higher quality mesh, especially at the boundaries, and retry. If there are still problems after this, consult the debugging options of the AD tool being used.

3. Once the tangent sensitivity is validated, construct and solve the adjoint algorithm, comparing the results with the validated tangent sensitivities. Numerical differences should be only due to rounding errors in floating-point arithmetic.

In the case that the tangent sensitivity results are incorrect after reducing the algorithm to its bare minimum (item 2, above), a useful test is to check whether or not the tangent algorithm is computing the primal results correctly, i.e. to check that the code was correctly interpreted by the AD tool. If it is the cas that the primal results are being incorrectly computed, the cause can be found by systematically replacing calls to derivative routines in the generated code with calls to the original routines until the problematic section is identified.

Duct flow surface sensitivity

Two cases are presented for an air duct from a VW Glof, a laminar case with $Re_{\mathrm{H}} = 60$, where H is the height of the duct at the inlet, and a "turbulent" case with $Re_{\mathrm{H}} = 600$ employing the Spalart-Allmaras turbulence model.[28] The standard wall function[29] is activated for $y^+ > 11.94$. In the latter case, a higher Reynolds number flow was attempted, but convergence to steady state was never achieved.

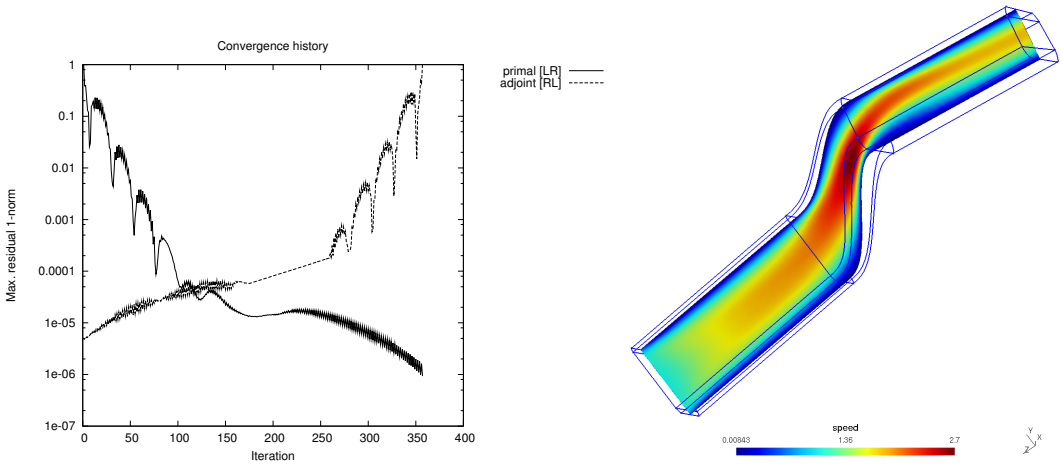
Flow field and sensitivity are computed using the in-house code GPDE because the ESI code is not yet ready to run sensitivity computations, however the in-house code does reflect accurately both in language and algorithm structure the ESI code being developed.

This case principally aims to demonstrate the capability of discrete adjoint algorithms. Here, the adjoint is generated from a spatially second order incompressible flow solver employing the SIMPLE scheme pressure-velocity coupling. Gradients are computed using the Green-Gauss theorem with two iterations and convection terms for the momentum are computed using the Van-Leer TVD scheme.[30]

Differentiation of the entire algorithm includes the mesh perturbation, nearest wall and turbulence modelling computation and a range of objective functions. The near wall model is differentiated transparently by the AD tool and by nature of the discrete approach the adjoint solver maintains the identical grid requirements. Average memory requirements are 1.7 times greater for the sensitivity computation compared to its primal and runtime is approximately trebled (one run of the primal plus one run of the adjoint which is twice the cost of the primal).

Convergence history (figure 8(a)) shows the primal being dropped five orders before the adjoint computation begins. The same number of iterations as the primal is applied to the adjoint. The is not desirable and could be omitted if there was a separation between the recording of flow logic and real valued data in adjoint source code and its retrieval.

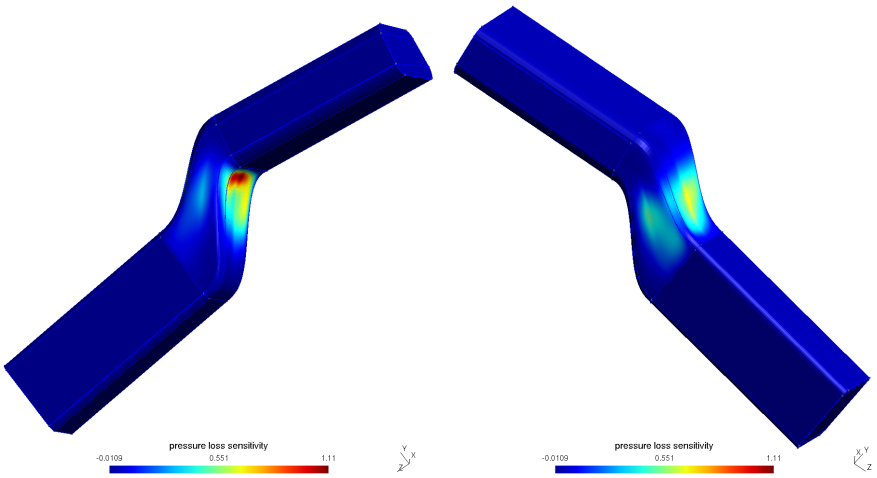
Surface sensitivity shown in figure is contour of the negated normalised pressure loss sensitivity w.r.t. the boundary vertex coordinates. Positive values indicate a mesh movement in the outward direction to lower the pressure loss.



(a) Convergence of primal and adjoint

(b) Fluid speed

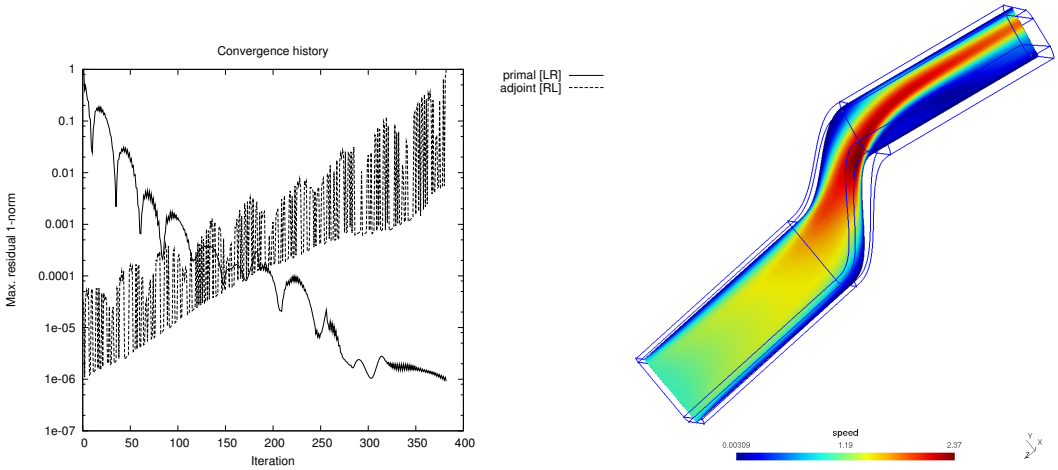
Figure 8. Duct flow case, $Re_{\mathrm{H}} = 60$



(a) Bottom view (b) Top view

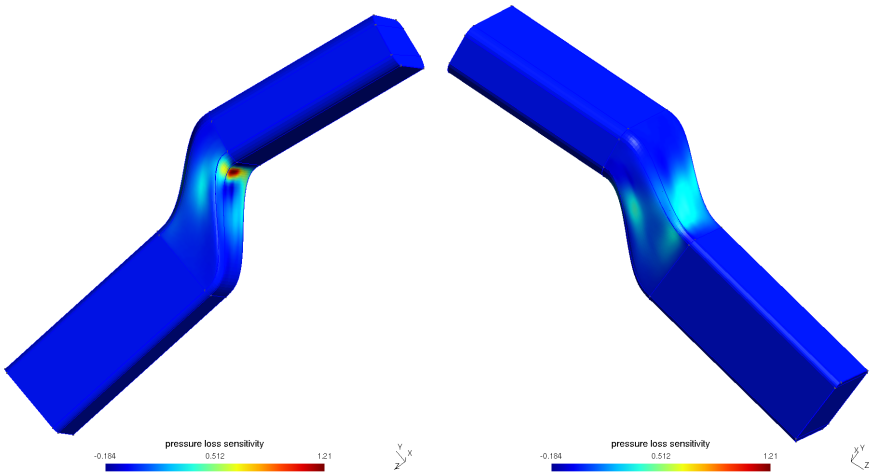
Figure 9. Duct pressure loss sensitivity w.r.t surface coordinates, $Re_{\mathrm{H}} = 60$

Increasing the Reynolds number and turning on the turbulence model presents no significant change in the surface sensitivity (figure 11). However, the relatively poor convergence of the adjoint (figure 10(a)) indicates than whilst the primal has shown reasonable convergence behaviour, reducing the norm six orders, it is likely that left for long enough the convergence would not reach a fixed point solution, i.e. the flow is unsteady. The short run-off downstream of the bend hinders the quality of the simulation, but this was the geometry provided in this work.



(a) Convergence of primal and adjoint (b) Fluid speed

Figure 10. Duct flow case, $Re_{\mathrm{H}} = 600$



(a) Bottom view (b) Top view

Figure 11. Duct pressure loss sensitivity w.r.t surface coordinates, $Re_{\mathrm{H}} = 600$

Conclusions

Development of a sensitivity algorithm using source code transformation at present requires considerable effort in preparing the source code: identifying the essential algorithm, introducing appropriate pragmas and conforming the source code to its language standard.

A good understanding of what advanced programming features are being used by the code is also important in order to find out whether or not differentiation is possible in the first place (or to what extent). A sure way of testing advanced language features with an AD tool is to perform it via a small example code, differentiating it in both forward and reverse mode and examining their behaviour.

Post-processing generated source code was performed using regular expression matching. This approach is barely acceptable, as it relies on a number of assumptions, and only parses broad features. A better solution would be to construct a processing utility which performs the preprocessing of the original code, calls the AD tool to generate the differentiated code and post-processes the generated code using an appropriate language parser. Such an arrangement would facilitate modification rules which require information about the original and generated code simultaneously.

This paper has aimed to outline one possible approach to systematically generating and validating adjoint code generation via source transformation and which facilities this work being done either in-house or outsourced. The demonstration implementation used here, GPDE, is available at `www.cfdpack.net/programs.html`.

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]Jameson, A., Martinelli, L., and Pierce, N., "Optimum Aerodynamic Design using the Navier-Stokes equations," *Theor. Comp. Fluid. Dyn.*, Vol. 10, 1998, pp. 213–237.

[2]Papadimitriou, D. I. and Giannakoglou, K., "A continuous adjoint method with objective function derivatives based on boundary integrals, for inviscid and viscous flows," *Computers Fluids*, Vol. 36, 2007, pp. 325–341.

[3]Othmer, C., Kaminski, T., and Giering, R., "Computation of Topological Sensitivities in Fluid Dynamics: Cost Function Versatility," *ECCOMAS CFD 2006*, edited by P. Wesseling, E. O. nate, and J. Périaux, TU Delft, 2006.

[4]Zymaris, A., Papadimitriou, D., Giannakoglou, K., and Othmer, C., "Continuous adjoint approach to the SpalartAllmaras turbulence model for incompressible flows," *Computers & Fluids*, Vol. 38, 2009, pp. 152838.

[5]Hovland, P., Mohammadi, B., and Bischof, C., "Automatic Differentiation of Navier-Stokes Computations," *Computational Methods for Optimal Design and Control*, edited by J. Borggaard, J. Burns, E. Cliff, and S. Schreck, Birkhäuser, Boston,

1998, pp. 265–284.

[6]Giles, M. B., Duta, M. C., and Müller, J.-D., "Adjoint Code Developments using the Exact Discrete Approach," *AIAA-CP-2001-2596*, 2001.

[7]Courty, F., Dervieux, A., Koobus, B., and Hascoët, L., "Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation," *Optimization Methods and Software*, Vol. 18, No. 5, 2003, pp. 615–627.

[8]Giles, M. B., Duta, M. C., Müller, J.-D., and Pierce, N. A., "Algorithm Developments for Discrete Adjoint Methods," *AIAA Journal*, Vol. 41, No. 2, 2003, pp. 198–205.

[9]Christakopoulos, F., Jones, D., and Mller, J.-D., "Pseudo-timestepping and verification for automatic differentiation derived CFD codes," *Computers and Fluids*, Vol. 46, No. 1, 2011, pp. 174 – 179, 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[10]Jones, D., Mller, J.-D., and Christakopoulos, F., "Preparation and assembly of discrete adjoint CFD codes," *Computers and Fluids*, Vol. 46, No. 1, 2011, pp. 282 – 286, 10th ICFD Conference Series on Numerical Methods for Fluid Dynamics (ICFD 2010).

[11]ESI, "Optimization using an Adjoint Method with PAM-FLOW," 2011, http://www.esi-cfd.com/content/view/751/205/, Accessed: 5 Dec 2011.

[12]Han, T., Hill, C., and Jindal, S., "Adjoint Method for Aerodynamic Shape Improvement in Comparison with Surface Pressure Gradient Method," *SAE Int. J. of Passeng. Cars Mech. Syst.*, Vol. 4, 2011, pp. 100–107.

[13]Bischof, C. H., Becker, H. M., Lang, B., Rasch, A., and Slusanschi, E., "Automatic Differentiation of Large-Scale Codes is no Illusion," Preprint BUW-SC 2005/3, Universität Wuppertal, 2005.

[14]Giering, R. and Kaminski, T., "Recipes for Adjoint Code Construction," *ACM Transactions on Mathematical Software*, Vol. 24, No. 4, 1998, pp. 437–474.

[15]autodiff.org, "Tools for Automatic Differentiation," http://www.autodiff.org/?module=Tools.

[16]Bendtsen, C. and Stauning, O., "FADBAD, a Flexible C++ Package for Automatic Differentiation," Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, aug 1996.

[17]Griewank, A., Juedes, D., and Utke, J., "Algorithm 755: ADOL-C: A Package for the Automatic Differentiation of Algorithms Written in C/C++," *ACM Transactions on Mathematical Software*, Vol. 22, No. 2, 1996, pp. 131–167.

[18]Pryce, J. D. and Reid, J. K., "ADO1, a Fortran 90 code for Automatic Differentiation," Tech. Rep. RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England, 1998.

[19]Naumann, U. and Riehme, J., "A Differentiation-Enabled Fortran 95 Compiler," *ACM Transactions on Mathematical Software*, Vol. 31, No. 4, 2005, pp. 458–474.

[20]Schlenkrich, S., Walther, A., Gauger, N., and Heinrich, R., "Differentiating Fixed Point Iterations with ADOL-C: Gradient Calculation for Fluid Dynamics," *Modeling, Simulation and Optimization of Complex Processes – Proceedings of 3rd HPSC 2006*, edited by H.-G. Bock, E. Kostina, H. Phu, and R. Rannacher, 2008, pp. 499 – 508.

[21]Stumm, P., Walther, A., Riehme, J., and Naumann, U., "Structure-Exploiting Automatic Differentiation of Finite Element Discretizations," *Advances in Automatic Differentiation*, edited by C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, and J. Utke, Springer, 2008, pp. 339–349.

[22]Aubert, P., Di Césaré, N., and Pironneau, O., "Automatic Differentiation in C++ Using Expression Templates and Application to a Flow Control Problem," *Computing and Visualization in Science*, Vol. 3, 2001, pp. 197–208.

[23]Giering, R., Kaminski, T., and Slawig, T., "Generating Efficient Derivative Code with TAF: Adjoint and Tangent Linear Euler Flow Around an Airfoil," *Future Generation Computer Systems*, Vol. 21, No. 8, 2005, pp. 1345–1355.

[24]Hascoët, L. and Pascual, V., "TAPENADE 2.1 User's Guide," Rapport technique 300, INRIA, Sophia Antipolis, 2004.

[25]Cusdin, P. and Müller, J.-D., "On the Performance of Discrete Adjoint CFD Codes using Automatic Differentiation," *International Journal of Numerical Methods in Fluids*, Vol. 47, No. 6-7, 2005, pp. 939–945, http://www.ea.qub.ac.uk/pcusdin.

[26]Chapman, S. J., *Fortran 95/2003 for scientists and engineers*, McGraw Hill, 3rd ed., 2008.

[27]Christianson, D., Davies, A., Dixon, L., Roy, R., and VanderZee, P., "Giving reverse differentiation a helping hand," *Optimization Methods & Software*, Vol. 8, No. 1, 1997, pp. 53–67, 2nd Internatioanal Workshop on Computational Differentiation, Santa Fe, NM, Feb, 1996.

[28]Spalart, P. R. and Allmaras, S. R., "A one-equation turbulence model for aerodynamic flows," *La Recherche Aerospatiale*, Vol. 1, No. 1, 1994, pp. 5–21.

[29]Ferziger, J. H. and Perić, M., *Computational Methods for Fluid Dynamics*, Springer, 3rd ed., 2002.

[30]Versteeg, H. K. and Malalasekera, W., *An Introduction to Computational Fluid Dynamics*, Pearson, 2nd ed., 2006.