

Experience with Automatic Differentiation of Hydra

David Radford
Cambridge University Engineering Department
CFD Laboratory

14th May 2004

AD Hydra

- Motivation
- Approach
- Experience
- Results

AD Hydra - Motivation

- Cascading of new features to linear and adjoint codes as non-linear solver is developed
 - Scheme updates
 - Turbulence models
 - Boundary conditions
 - Complex objective functions for turbomachinery
- Improved consistency and accuracy of codes

Outline of AD Method

- Tapedcode adopted as AD code
 - Free for academic use
 - Runs locally
 - Oxford experience
- Existing framework of linear/adjoint solvers retained
- AD subroutines substituted for hand-coded ones

AD Application Method

- Method adopted was not strictly automatic
- Hydra flux routines contain
 - Wrapping parallel access calls
 - Loops over edges or nodes to accumulate fluxes
- Tapenade treats parallel calls as inaccessible subroutines and adds in corresponding calls
- In reverse mode, loops lead to unnecessary storage through *push* and *pop* functions – potentially expensive in memory and CPU terms

AD Application Method

- For routines containing loops over edges (viscous and inviscid fluxes)
 - *per edge* “work” part of routine split into a new subroutine
 - New subroutine called for each edge and passed data for edge and end nodes only
 - Tapenade applied to *per edge* subroutine, avoiding parallel calls and the need for most *push/pop* calls
- Method suggested by Cusdin – found to give ~80% reduction in run time

AD Application Method

- For routines containing loops over *nodes* (source terms), “brute force” approach was used
- Parallel calls commented out, and Tapenade applied to whole subroutine
- For adjoint, produces separate forward and reverse loops

AD Application Method

- In both cases, AD output was modified by hand
 - Unnecessary non-linear calculations removed
- Much to do in reverse mode
 - Merging of forward/reverse loops
 - Branch constructs replaced with original tests or saved “sign” variables
 - Removal of *push/pop* calls – most unnecessary, some replaced by declared variables

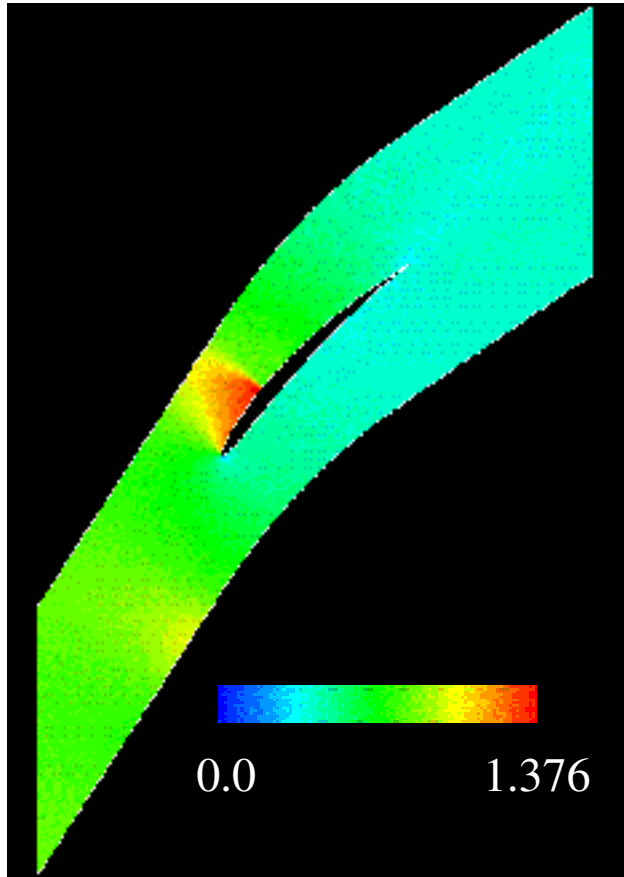
Pitfalls...

- If a low-level subroutine is sent to Tapenade as the “head” for differentiation, the AD code includes initialisation to zero of incoming variables, often hidden in the middle of the routine
 - This disrupts accumulation of fluxes in solver
 - Zeroing loops need to be removed and variables incremented rather than assigned a value first time

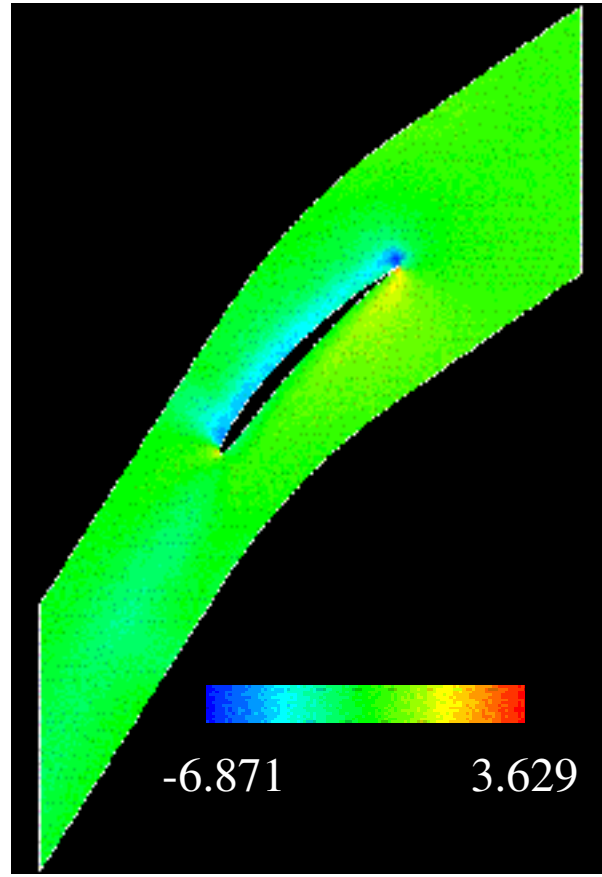
Pitfalls...

- Careful testing needed
 - *Testlin* and *testadj* programs used – compare linear code against complex non-linear, then adjoint against linear
 - Testing of individual and combined subroutines both highlight problems
 - The test programs and their components have to be up to date too!

Testcase - 10th Standard Configuration Compressor



Mach Number Contours

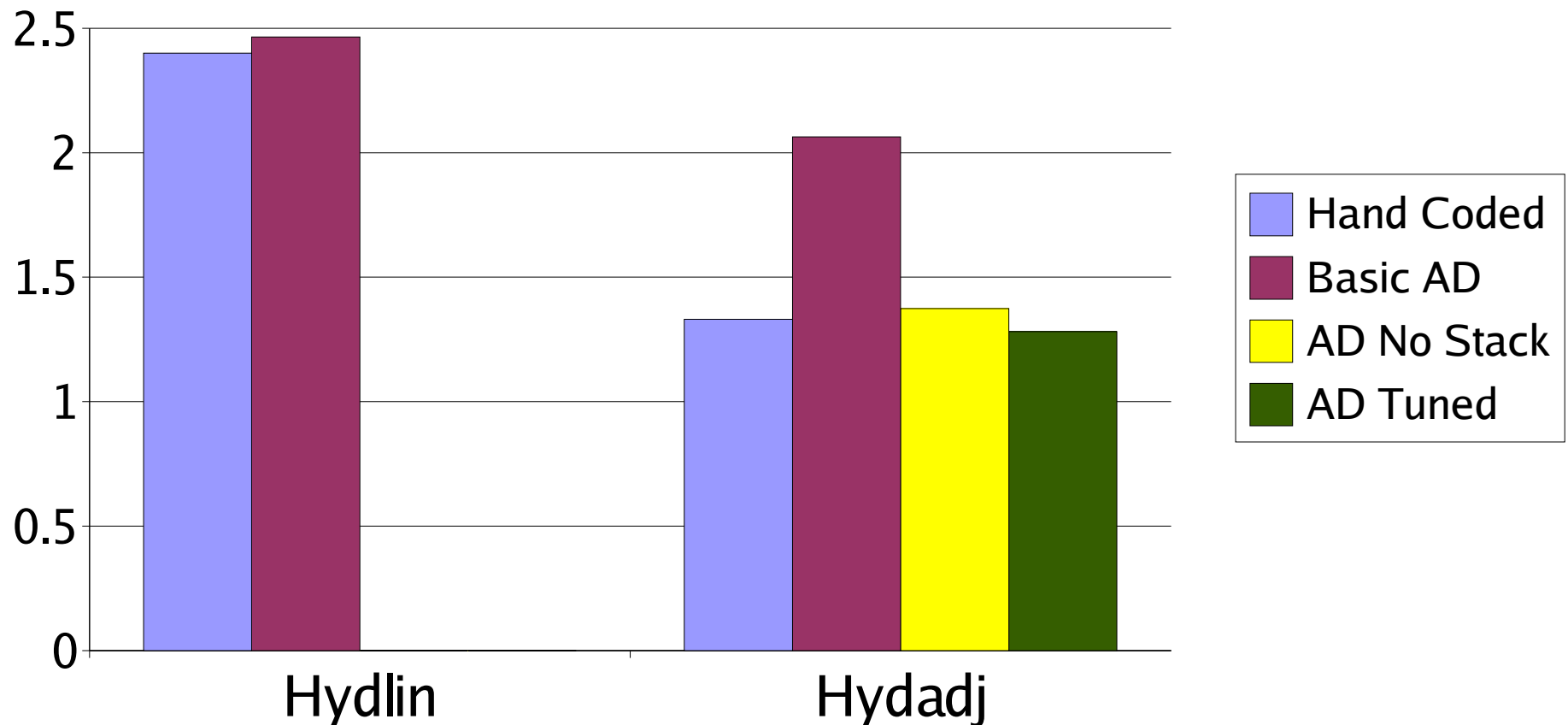


2nd (Axial Momentum)
Adjoint Contours for
Axial Surface Force
ObjectiveFunction

- Inviscid
- 6% thick 2D compressor section
- ~2900 node triangular unstructured mesh
- $M_{inlet} = 0.8$, $M_{exit} = 0.45$, transonic over blade
- Design variable is anti-clockwise rotation
- Objective functions: Mass flow, exit angle, loss, axial and tangential forces

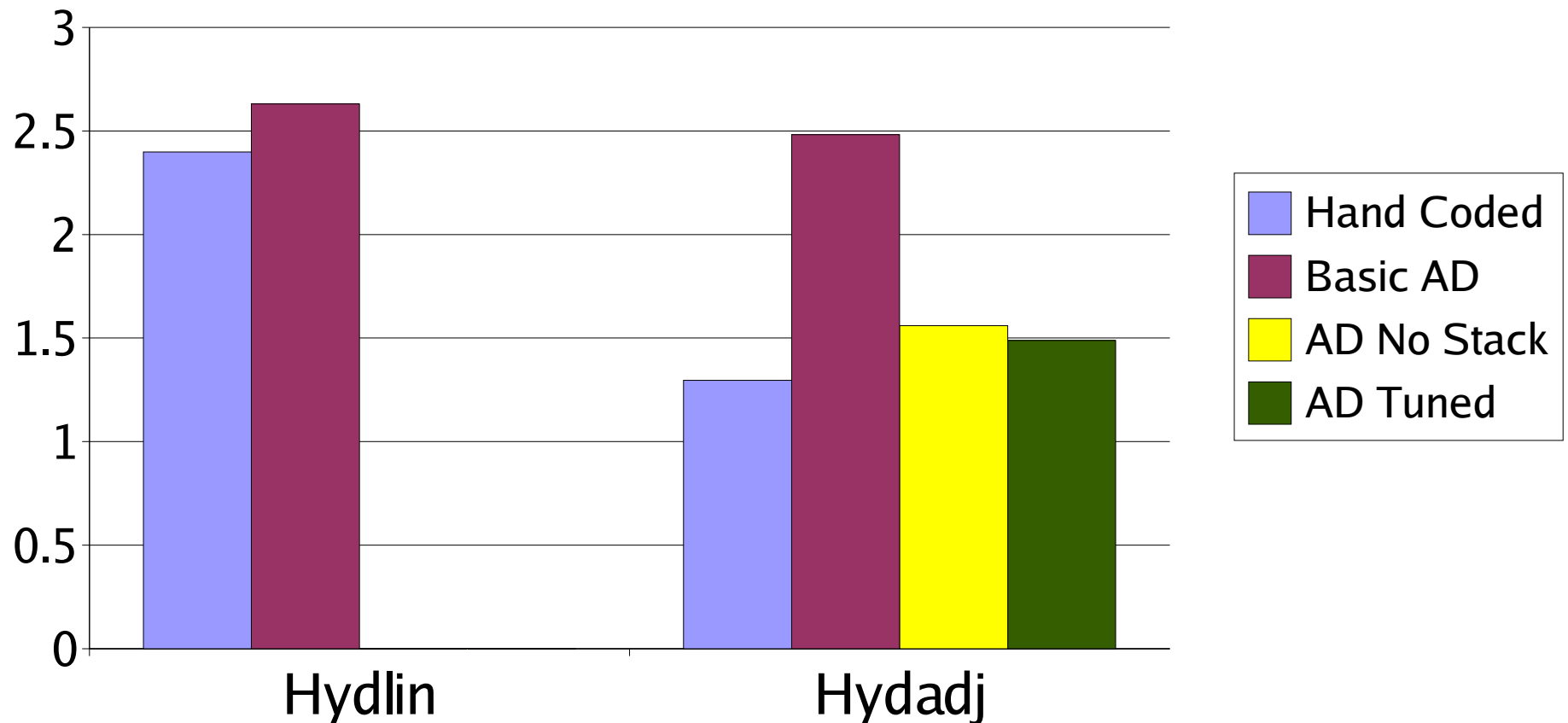
Runtime Comparisons – P3, Absoft

Comparison of CPU time for 300 iterations, Normalised Against Non-Linear Solver, for Hand-Coded and AD-based Hydra Linear and Adjoint solvers



Runtime Comparisons – P4M, Intel

Comparison of CPU time for 300 iterations, Normalised Against Non-Linear Solver, for Hand-Coded and AD-based Hydra Linear and Adjoint solvers



Sensitivity Comparisons

Objective Function	Sensitivities to Anti-Clockwise Rotation (per radian)				
	Finite Difference	Hand Coded	Linear/FD Factor	AD	Linear/FD Factor
Mass Flow Rate	-0.052301	-0.057452	1.098491	-0.055077	1.053087
Total Pressure Loss (%)	-16.303193	-15.908880	0.975814	-16.199388	0.993633
Squared Angle	1.513669	1.506837	0.995486	1.517176	1.002317
Axial Force	0.222551	0.244486	1.098562	0.235225	1.056948
Tangential Force	-0.348742	-0.362438	1.039271	-0.358393	1.027673

- Sensitivities to anti-clockwise rotation of blade section shown for original hand-coded and AD linear/adjoint solvers (linear and adjoint values the same). Finite difference values calculated using +/- 0.001 degree perturbations with central differencing.
- Consistent AD codes show improvement in accuracy vs FD
- Still room for improvement

Areas for Further Development

- Adjoint pre-processor
 - limited use has been made of AD for awkward objective functions such as blockage (based on max/min of flow data) and flux averaged quantities (complicated averaging)
- Boundary conditions
 - Perturbation quantities in linear code not direct differentials of equivalents in non-linear code

AD Linear Pre-Processor

- Linear pre-processor *rhslin* calculates linear forcing term $\partial R / \partial \alpha$ for input boundary node displacement sensitivities $\partial x_b / \partial \alpha$
- Complex versions of edge weight and flux calculations used to propagate sensitivities from boundaries into mesh and to flow
- Complex variables not supported by current parallel library – limiting mesh size
- Using AD, all calculations can be achieved using real data types, removing mesh size restriction

AD Reverse Linear Pre-Processor

- Currently, the linear pre-processor takes boundary coordinate sensitivities and propagates these through to residual sensitivities
- Functional sensitivities given by dot product with adjoint variables
- With AD, a fully reversed process becomes possible
- Takes adjoint flow variables, and works back to adjoint mesh sensitivities

AD Reverse Linear Pre-Processor

- Process only needs to be done once per objective function – eliminates need for multiple mesh movement solutions
- Objective function sensitivities will be produced by dot products over boundary nodes only

$$\begin{aligned} v^T \left(\frac{\partial R}{\partial \alpha} \right) &= v^T \left(\frac{\partial R}{\partial X} \right) \left(\frac{\partial X}{\partial x_B} \right) \left(\frac{\partial x_B}{\partial \alpha} \right) \\ &= \left(\frac{\partial x_B}{\partial \alpha} \right)^T \left[\left(\frac{\partial X}{\partial x_B} \right)^T \left(\frac{\partial R}{\partial X} \right)^T v \right]^T \end{aligned}$$

Summary

- Application of AD to Hydra is a practical means of keeping the code up to date and expanding the range of functionality
- Structure of non-linear code can be altered to improve output
- Hand-tuning still required for performance and compatibility with solver and programming framework
- Application to codes outside core solver offers future improvements in adjoint applications