Discrete Adjoint of the Incompressible Naiver-Stokes Equations

D. P. Jones, J.-D. Müller, J. Riehme
Queen Mary, University of London
University of Hertfordshire

December, 2010

Overview

- As part of a European project, the discrete adjoint of a commercial CFD solver is being developed using automatic differentiation (AD).
- The use of AD potentially holds the promise of delivering maintainable adjoint codes for complex algorithms.
- However it remains to be demonstrated that current AD tools are indeed capable of deriving efficient adjoint codes which contain both significant programming and algorithmic complexity.

Specification I

- Discrete differentiation of the sparse linear solver ought not be necessary.
- The recording of variables in reverse mode should be minimised by identifying iteratively independent loops.
- High level language programming features should be preserved.
- Building the program should be easily defined in a Makefile.
- Constructing variations of the differential solver for validation purposes should be possible via the build process.
- The complete package ought to be maintainable by the original code developer, rather than the adjoint algorithm developer.

Specification II

- The means of generating the optimised adjoint algorithm should be as independent from the AD tool as possible (i.e. regarding external function differentiation).
- Whatever code preparation needs to be done, the task should require no understanding of adjoints. The essential requisites for this kind of code preparation are scripting with regular expressions and some insight to the code algorithm.
- Maintenance of the optimised adjoint algorithm will be required but this will require only minimal insight of discrete adjoints. Furthermore, its maintenance will be aided by the generated brute-force adjoint produced by the AD tool.

Approach

1. File naming
2. Macro processing
3. Transformed code modifications
4. Subroutine redirection
5. Determining the compilation order
6. Fixed point adjoint algorithm

Top level differentiation

```
subroutine state_objective(state, ..., obj)

  do iter=1, ...
    call mesh_perturbation(state, ...)
  end do

  call geometry(...)

  do iter=1, ...
    call naiver_stokes(...)
  end do

  call cost_function(..., obj)
end subroutine
```

Figure: Top level differentiation: input: `state`, output: `obj`

Macro processing

```
m4_include(definitions.m4)
module solve_m
contains

M4_ASSERT(`M4_MATCH(`RETAIN',ALL) || M4_MATCH(`RETAIN',ACTIVE)',``
  subroutine solve(a, x, b)
    !... LINEAR EQUATIONS SOLVER
  end subroutine
'')

M4_ASSERT(`M4_MATCH(`AD_MODE',FWD) && M4_MATCH(`RETAIN',PASSIVE)',``
  subroutine solve_d(a, ad, x, xd, b, bd)
    !... MANUALLY FORMULATED TANGENT
  end subroutine
'')

M4_ASSERT(`M4_MATCH(`AD_MODE',REV) && M4_MATCH(`RETAIN',PASSIVE)',``
  subroutine solve_b(a, ab, x, xb, b, bb)
    !... MANUALLY FORMULATED ADJOINT
  end subroutine
'')

end module
```

Figure: Using the manually differentiated solvers

Makefile structure

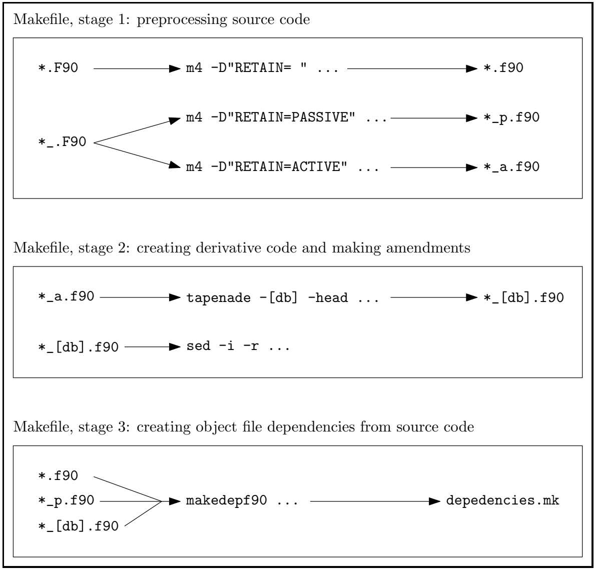

Makefile, stage 1: preprocessing source code
*.F90
m4 -D"RETAIN= " ...
*.f90
*_.F90
m4 -D"RETAIN=PASSIVE" ...
*_p.f90
m4 -D"RETAIN=ACTIVE" ...
*_a.f90
Makefile, stage 2: creating derivative code and making amendments
*_a.f90
tapenade -[db] -head ...
*_[db].f90
*_[db].f90
sed -i -r ...
Makefile, stage 3: creating object file dependencies from source code
*.f90
*_p.f90
*_[db].f90
makedepf90 ...
depedencies.mk

Figure: Preprocessing, differentiating and dependency order

Using UMFPACK

```
do iter=1,n_iter
  id=1
  call momentum_eqn(...)
  if(last_iter)then
    call umf4sym(id,...)
    call umf4num(id,...)
  end if

  id=2
  call continuity_eqn(...)
  if(last_iter)then
    call umf4sym(id,...)
    call umf4num(id,...)
  end if

  if(resid<cutoff)exit
end do
```

(a) Primal iterative loop

```
subroutine linear_eqn_solver_b (id, ...)
  norm = p_norm (phib-matvec_prod(mat, rhsb), 1)

  sys=1  ! transpose matrix
  call umf4sol (sys, rhsb, phib, id)

  phib = 0

  do i=1,size(ia)-1
    do k=ia(i),ia(i+1)-1
      j=ja(k)
      matb(k) = matb(k) - rhsb(i) * phi(j)
    end do
  end do
end subroutine
```

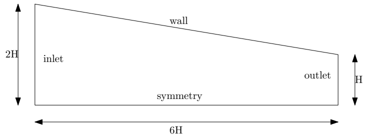
(b) Alternative adjoint linear solver routine

Figure: Primal and adjoint modifications

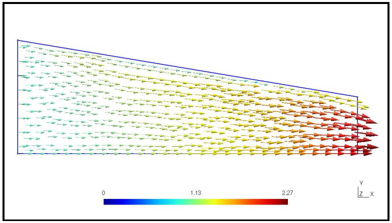
Problems

1. Not using preconditioning in the linear solver.
2. Not converging the system of equations $A^T\bar{b} = \bar{\phi}$ sufficiently.
3. Using elaborate array indexing, such as `vel(offset(:)+i)`.
4. Not making local copies of global data (especially for boundary conditions).
5. Passing instances of derived data types to subroutines several levels deep.
6. Having aliased function calls, such as `x = f(x)`.
7. Indicating iteratively independent loops when they are not.
8. Not defining local array sizes of adjoint variables based on their primal variables.

Results I



(a) Geometry and BCs



(b) Flow field

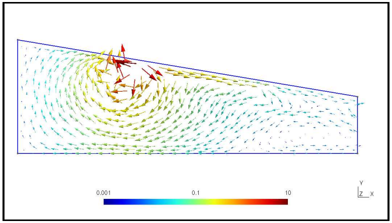
Figure: Nozzle geometry and flow field

Results II

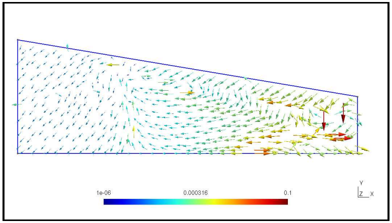
Table: Nozzle flow sensitivity validation

	Sensitivity
FD	-31.9485227732451
Tangent	-31.9544194402137
Adjoint	-31.9544194388609

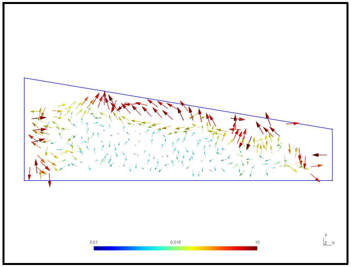
Results III



(a) $\frac{\partial u}{\partial x}$ via tangent mode



(b) FD sensitivity error



(c) $\frac{\partial F_{wall,y}}{\partial x}$ via adjoint mode

Desirable AD tool features

- Automatically identifying iteratively independent loops.
- Filtering source code into passive and active code, i.e. split input files after parsing top-level dependencies.
- Supporting a fixed-point solver pragma.
- Supporting a pragma to denote that a subroutine has an interface (or realise it belongs to a module).

Summary and further work

- A robust approach to constructing the adjoint of fixed-point coupled PDE solvers has been demonstrated, yielding an adjoint to primal run-time ratio of approximately 2.3.
- Work has already begun by Jan Riehme to generate the adjoint via operator overloading approach using the NAG CompAD tool.
- Collaboration with Tapenade and NAG CompAD developers is on-going.

A case for D in AD

The D programming language appears to strongly lend itself to source code transformation:

- Parsing is done in one pass (compilation is very fast),
- Hash tables and arrays are built-in types,
- Multi-threading support built in,
- Code can be modulated (like F90); header files are not necessary,
- The pre-processor built into the language, unlike C, etc.
- Supports C style code, OO, scripting, templating, functional programming, Regex
- Proper error handling
- Easy to pick up from a Fortran background

Queen Mary
University of London

www.gpde.net

esi
get it right®

FLOWHEAD
Fluid Optimisation Workflows for Highly Effective Automotive Development Processes