Using Automatic Differentiation for Adjoint CFD Code Development

Mike Giles  Devendra Ghate  Mihai Duta

Oxford University Computing Laboratory
Overview

- why discrete adjoint?
- automatic differentiation
- an airfoil code example
Discrete Adjoint

Adjoint methods are very efficient for obtaining the sensitivity of one output (objective function) to many inputs (design parameters).

There are two adjoint approaches:

- **continuous**: construct adjoint PDE and then discretise.
- **discrete**: discretise original nonlinear PDE, then linearise and use its adjoint/transpose.

Both approximate the true gradient of the output – latter gives the gradient of the discrete output but this consistency is unnecessary for many optimisers.
Discrete Adjoints

I prefer discrete adjoints because:

- clear prescriptive process for constructing the discrete adjoint equations and boundary conditions;
- usually guaranteed to get same iterative convergence rate as original nonlinear code;
- Automatic Differentiation can simplify the development of the adjoint CFD code.
Discrete Adjoint

Suppose an input $\alpha$ leads to an output $J$:

$$\alpha \rightarrow X \rightarrow U \rightarrow J.$$ 

Defining $\dot{\alpha}, \dot{X}, \dot{U}, \dot{J}$ to be derivatives with respect to $\alpha$,

$$\dot{X} = \frac{\partial X}{\partial \alpha} \dot{\alpha}, \quad \dot{U} = \frac{\partial U}{\partial X} \dot{X}, \quad \dot{J} = \frac{\partial J}{\partial U} \dot{U},$$

$$\Rightarrow \dot{J} = \frac{\partial J}{\partial U} \frac{\partial U}{\partial X} \frac{\partial X}{\partial \alpha} \dot{\alpha},$$

Note this calculation goes forward:

$$\dot{\alpha} \rightarrow \dot{X} \rightarrow \dot{U} \rightarrow \dot{J},$$
Discrete Adjoints

Now, defining $\overline{\alpha}, \overline{X}, \overline{U}, \overline{J}$ to be derivatives of $J$ with respect to $\alpha, X, U, J$,

$$\overline{\alpha} \overset{\text{def}}{=} \left( \frac{\partial J}{\partial \alpha} \right)^T = \left( \frac{\partial J}{\partial X} \frac{\partial X}{\partial \alpha} \right)^T = \left( \frac{\partial X}{\partial \alpha} \right)^T \overline{X},$$

and similarly $\overline{X} = \left( \frac{\partial U}{\partial X} \right)^T \overline{U}, \quad \overline{U} = \left( \frac{\partial U}{\partial J} \right)^T \overline{J}$,

$$\implies \overline{\alpha} = \left( \frac{\partial X}{\partial \alpha} \right)^T \left( \frac{\partial U}{\partial X} \right)^T \left( \frac{\partial U}{\partial J} \right)^T \overline{J}.$$

Note this calculation goes backward:

$$\overline{\alpha} \leftarrow \overline{X} \leftarrow \overline{U} \leftarrow \overline{J}.$$
Automatic Differentiation

AD views each floating point operation as a separate stage:

Note requirement to store data on forward pass in order to use partial derivatives on reverse pass.
Automatic Differentiation

To create key parts of our linear and adjoint CFD codes, we use AD software called Tapenade:

- developed at INRIA by Laurent Hascœet and Valerie Pascual
- uses source code transformation, takes a Fortran subroutine as input and generates a new Fortran subroutine as output
- given a subroutine which computes \( f(u) \), it can create routines to evaluate
  
  \[ \dot{f} = \frac{\partial f}{\partial u} \dot{u} \]  
  (forward mode)
  
  \[ \overline{u} = \left( \frac{\partial f}{\partial u} \right)^T \overline{f} \]  
  (reverse mode)
Discrete Adjoint

Steady discrete CFD equations

\[ N(U, X) = 0. \]

are often solved by iteration

\[ U^{n+1} = U^n - P(U^n, X) N(U^n, X), \]

The linearised equations

\[ L \dot{U} + \dot{N} = 0, \quad L \equiv \frac{\partial N}{\partial U}, \quad \dot{N} = \frac{\partial N}{\partial X} \dot{X}, \]

can then be solved by the iteration

\[ \dot{U}^{n+1} = \dot{U}^n - P \left( L \dot{U}^n + \dot{N} \right). \]
Discrete Adjoints

Since

$$\dot{U} = - L^{-1} \dot{N},$$

the adjoint equation is

$$\overline{N} = - (L^T)^{-1} \overline{U} \quad \implies \quad L^T \overline{N} + \overline{U} = 0,$$

which can be solved iteratively using

$$\overline{N}^{n+1} = \overline{N}^n - P^T \left( L^T \overline{N}^n + \overline{U} \right).$$

Written paper explains that $P^T$ corresponds to same time-marching algorithm in simple cases, and we use AD to get code to evaluate $L^T \overline{N}$ and $\overline{U}$ (and also $LU$ and $\dot{N}$ for linear code).
Airfoil Code

Very simple 2D airfoil code:
- cell-centred, unstructured quadrilateral grid
- inviscid fluxes plus simple numerical smoothing
- simple predictor/corrector time-marching with local timesteps

Starting with a nonlinear solver, we use AD to generate the key bits of both linear and adjoint codes.
Airfoil Code

Fortran files:

- airfoil.F
  nonlinear code

- air_lin.F
  linear code

- air_adj.F
  adjoint code

- testlinadj.F
  validation code

- input.F
  input/output routines

- routines.F
  nonlinear routines
Airfoil Code

Nonlinear routines:

- **TIME_CELL**: computes the local area/timestep for a single cell
- **FLUX_FACE**: computes the flux through a single regular face
- **FLUX_WALL**: computes the flux for a single airfoil wall face
- **LIFT_WALL**: computes lift contribution from a single airfoil wall face
Airfoil Code

```fortran
#ifdef COMPLEX
   subroutine Cflux_wall(x1,x2,q,res)
#else
   subroutine flux_wall(x1,x2,q,res)
#endif

c compute momentum flux from an individual wall face
implicit none
#include "const.inc"
integer n
#ifdef COMPLEX
   complex*16 & x1(2),x2(2),q(4),res(4),
#else
   real*8 & x1(2),x2(2),q(4),res(4),
\& dx,dy, ri,u,v,p
c
dx = x1(1) - x2(1)
dy = x1(2) - x2(2)
c
ri = 1.d0/q(1)
u = ri*q(2)
v = ri*q(3)
p = gm1*(ri*q(4) - 0.5*(u**2+v**2))
c
res(2) = res(2) - p*dy
res(3) = res(3) + p*dx
c
return
end
```
Airfoil Code

Different AD-generated versions of wall flux routine:

FLUX_WALL(x1, x2, q, res)

FLUX_WALL_D(x1, x2, q, qd, res, resid)

FLUX_WALL_DX(x1, x1d, x2, x2d, q, qd, res, resid)

FLUX_WALL_B(x1, x2, q, qb, res, resb)

FLUX_WALL_DX(x1, x1b, x2, x2b, q, qb, res, resb)
Part of Makefile:

```
flux_wall_d.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -forward \
        -head       flux_wall \
        -output     flux_wall \n\n-flux_wall_d.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -forward \n        -head       flux_wall \n        -output     flux_wall \n        -vars       "q res" \n        -outvars    "q res" \n        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_d.f;
/bin/rm routines.f flux_wall_d.f *.msg
```

```
flux_wall_dx.o: routines.F
    ${GCC} -E -C -P routines.F > routines.f;
    ${TPN} -forward \n        -head       flux_wall \n        -output     flux_wall \n        -vars       "x1 x2 q res" \n        -outvars    "x1 x2 q res" \n        -difffuncname "_dx" \n        routines.f;
    ${FC} ${FFLAGS} -c flux_wall_dx.f;
/bin/rm routines.f flux_wall_dx.f *.msg
```
Airfoil Code

Part of Makefile:

\[ \text{flux\_wall\_b.o: routines.F} \]
\[ \{\text{GCC}\} \ -E \ -C \ -P \ \text{routines.F} > \ \text{routines.f}; \]
\[ \{\text{TPN}\} \ -\text{backward} \]
\[ \quad -\text{head} \quad \text{flux\_wall} \]
\[ \quad -\text{output} \quad \text{flux\_wall} \]
\[ \quad -\text{vars} \quad "q \ res" \]
\[ \quad -\text{outvars} \quad "q \ res" \]
\[ \quad \text{routines.f}; \]
\[ \{\text{FC}\} \ \{\text{FFLAGS}\} \ -c \ \text{flux\_wall\_b.f}; \]
\[ /\text{bin}/\text{rm} \ \text{routines.f} \ \text{flux\_wall\_b.f} \ *.\text{msg} \]

\[ \text{flux\_wall\_bx.o: routines.F} \]
\[ \{\text{GCC}\} \ -E \ -C \ -P \ \text{routines.F} > \ \text{routines.f}; \]
\[ \{\text{TPN}\} \ -\text{backward} \]
\[ \quad -\text{head} \quad \text{flux\_wall} \]
\[ \quad -\text{output} \quad \text{flux\_wall} \]
\[ \quad -\text{vars} \quad "x1 \ x2 \ q \ res" \]
\[ \quad -\text{outvars} \quad "x1 \ x2 \ q \ res" \]
\[ \quad -\text{difffuncname} \quad "\_bx" \]
\[ \quad \text{routines.f}; \]
\[ \{\text{FC}\} \ \{\text{FFLAGS}\} \ -c \ \text{flux\_wall\_bx.f}; \]
\[ /\text{bin}/\text{rm} \ \text{routines.f} \ \text{flux\_wall\_bx.f} \ *.\text{msg} \]
Nonlinear Code

define grid and initialise flow field

begin predictor/corrector time-marching loop
    loop over cells to calculate timestep
        call TIME_CELL
    loop over regular faces to calculate flux
        call FLUX_FACE
    loop over airfoil faces to calculate flux
        call FLUX_WALL
    loop over cells to update solution
end time-marching loop

calculate lift
    loop over boundary faces
        call LIFT_WALL
define grid and initialise flow field
define grid perturbation

loop over cells -- perturbed timestep
  call TIME_CELL_DX
loop over regular faces -- perturbed flux
  call FLUX_FACE_DX
loop over airfoil faces -- perturbed flux
  call FLUX_WALL_DX

begin predictor/corrector time-marching loop
  loop over cells to calculate timestep
    call TIME_CELL_D
  loop over regular faces to calculate flux
    call FLUX_FACE_D
  loop over airfoil faces to calculate flux
    call FLUX_WALL_D
  loop over cells to update solution
end time-marching loop

calculate lift
  loop over boundary faces -- perturbed lift
    call LIFT_WALL_DX
define grid and initialise flow field

calculate adjoint lift sensitivity
  loop over boundary faces
    call LIFT_WALL_BX

begin predictor/corrector time-marching loop
  loop over airfoil faces -- adjoint flux
    call FLUX_WALL_B
  loop over regular faces -- adjoint flux
    call FLUX_FACE_B
  loop over cells -- adjoint timestep calc
    call TIME_CELL_B
  loop over cells to update solution
end time-marching loop

loop over airfoil faces -- adjoint flux
  call FLUX_WALL_BX
loop over regular faces -- adjoint flux
  call FLUX_FACE_BX
loop over cells -- adjoint timestep calc
  call TIME_CELL_BX

loop over nodes to evaluate lift sensitivity
Validation is very important – can be done at various levels starting with the consistency of the individual nonlinear, linear and adjoint routines.

The “complex variable trick” gives

\[
\lim_{\epsilon \to 0} \frac{i}{\epsilon} \mathcal{T} \{ f(u + i\epsilon \dot{u}) \} = \frac{\partial f}{\partial u} \dot{u}
\]

so the linear code can be checked against the nonlinear.

Also, by taking different unit vectors \( \dot{u} \) can get \( \frac{\partial f}{\partial u} \), and compare to \( \left( \frac{\partial f}{\partial u} \right)^T \) from adjoint routine.
Airfoil Code

The next level is iterative equivalence between the linear and adjoint codes.

If both codes are initialised to zero ($\dot{U}^0 = \overline{N}^0 = 0$) then the paper explains that

$$\underbrace{(\overline{N}^n)^T \dot{N}}_{\text{adjoint code}} = \underbrace{\overline{U}^T \dot{U}^n}_{\text{linear code}}$$

i.e. after the same number of iterations $n$, both codes should give the same value for the linear sensitivity of the output (lift) to one input (angle of attack).
Airfoil Code

Final check is between linear sensitivity and nonlinear finite difference

\[ O(\Delta \alpha^2) \] error due to finite difference step size \( \Delta \alpha \)

\[ O(\varepsilon / \Delta \alpha) \] error due to finite machine precision \( \varepsilon \)
Conclusions

I think discrete adjoints are preferable to continuous adjoints for practical reasons:
- clear prescriptive process;
- same iterative convergence rate;
- AD can simplify code development;

AD tools are now quite mature, and I personally recommend Tapenade.

Validation is also very important and can be done at a number of different levels to give confidence in the adjoint code.

Finally, there will be a “hands-on” tutorial at ADA on Wednesday.