Abstract

Adjoint code for 1-D and 2-D Euler equations are developed using automatic differentiation tool called Tapenade. A piecemeal approach is used in which the subroutines in the flow solver are differentiated individually and used in an adjoint iterative solver. This approach is useful for problems requiring iterative solution procedures since it leads to enormous savings in memory and time. For 2-D case, the adjoint solver requires about 38% more memory compared to the flow solver. The time per adjoint iteration is about twice that of the flow solver. The adjoint code is used to solve pressure matching problem for quasi 1-D flow through a duct. A smoothing procedure based on an elliptic equation is developed for this purpose. In 2-D, a second order vertex-centroid scheme on triangular grids is used to develop an adjoint solver. Both the flow and adjoint solvers are accelerated using LUSGS scheme with spectral radius approximation for flux jacobians. The adjoint code is validated by computing the slope of the $C_l - \alpha$ curve.

Contents

.

1 Introduction

Consider the problem of minimizing a given *cost function* J which depends on some *control/design variable* α and *state variable* u, i.e., $J \equiv J(\alpha, u)$. The state variable is itself constrained to satisfy a *state equation* $R(\alpha, u) = 0$, which gives an implicit dependence of u on α.

$$\min_{\alpha} J(\alpha, u) \quad \text{s.t.} \quad R(\alpha, u) = 0 \tag{1}$$

In the case of shape optimization of a flow system, the state equation could be Euler or Navier-Stokes equations and α is a set of control variables which parameterize the shape. Practical methods of solving this problem are iterative in nature; an initial guess is made for α which is iteratively corrected until some convergence criterion is satisfied.

In gradient-based optimization techniques like steepest descent method, the correction to the control variable is made using the gradient of the cost function. If the control variable changes by an amount $\delta\alpha$, then the change in the cost function, to first order is

$$\delta J = \frac{\partial J}{\partial \alpha}\delta\alpha + \frac{\partial J}{\partial u}\delta u = \left(\frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial u}\frac{\partial u}{\partial \alpha}\right)\delta\alpha$$

or introducing the gradient

$$G := \frac{\partial J}{\partial \alpha} + \frac{\partial J}{\partial u}\frac{\partial u}{\partial \alpha}$$

we have $\delta J = G\delta\alpha$. The cost function will decrease if we choose $\delta\alpha = -\epsilon G^{\mathrm{T}}$ for some $\epsilon > 0$, since $\delta J = -\epsilon G^{\mathrm{T}} G = -\epsilon \|G\|^2 \leq 0$. But the change in the state variable δu or the sensitivity $\frac{\partial u}{\partial \alpha}$ must be obtained from the state equation. Linearizing the state equation we obtain

$$\frac{\partial R}{\partial \alpha}\delta\alpha + \frac{\partial R}{\partial u}\delta u = 0 \quad \Longrightarrow \quad \boxed{\frac{\partial R}{\partial u}\frac{\partial u}{\partial \alpha} = -\frac{\partial R}{\partial \alpha}}$$

The above sensitivity equation is usually very large and has to be solved iteratively. Introducing a pseudo-time derivative

$$\frac{\partial}{\partial t}\frac{\partial u}{\partial \alpha} + \frac{\partial R}{\partial u}\frac{\partial u}{\partial \alpha} = -\frac{\partial R}{\partial \alpha}$$

the above equation can be solved using local time-stepping combined with an implicit scheme for convergence acceleration. Note that the sensitivity equations have the same eigenvalues as the primal problem $R = 0$ and we can expect similar convergence characteristics for both equations. In many practical applications the number of design variables is large; since each design variable leads to one sensitivity equation, the total cost in obtaining the gradient can be quite large. The cost can be reduced by using the *adjoint method*. Introducing the *adjoint variable* v we can write the change in cost function as

$$\begin{aligned} \delta J &= \frac{\partial J}{\partial \alpha}\delta\alpha + \frac{\partial J}{\partial u}\delta u + v^{\mathrm{T}}\left(\frac{\partial R}{\partial \alpha}\delta\alpha + \frac{\partial R}{\partial u}\delta u\right) \\ &= \left(\frac{\partial J}{\partial \alpha} + v^{\mathrm{T}}\frac{\partial R}{\partial \alpha}\right)\delta\alpha + \left(\frac{\partial J}{\partial u} + v^{\mathrm{T}}\frac{\partial R}{\partial u}\right)\delta u \end{aligned}$$

The dependence on the state variable or sensitivity can be removed by setting the coefficient of δu to zero

$$\frac{\partial J}{\partial u} + v^{\mathrm{T}} \frac{\partial R}{\partial u} = 0 \quad \Longrightarrow \quad \boxed{\left(\frac{\partial R}{\partial u}\right)^{\mathrm{T}} v = -\left(\frac{\partial J}{\partial u}\right)^{\mathrm{T}}}$$

which is called the *adjoint equation*. The adjoint equation also has the same eigenvalues as the sensitivity equation and is solved iteratively by introducing a pseudo-time term. Note that the adjoint equation does not have any derivatives with respect to the control variables. Hence the cost of solving the adjoint equations is nearly independent of the number of control variables. Once the adjoint solution has been obtained the gradient can be computed from

$$G = \frac{\partial J}{\partial \alpha} + v^{\mathrm{T}} \frac{\partial R}{\partial \alpha}$$

which is cheaper to evaluate since it does not require any iterative solution and involves matrix-vector products.

The sensitivity and adjoint equations contain derivatives of J and R. These can be evaluated in several ways.

1. Hand differentiation
2. Finite difference method
3. Complex variable method
4. Automatic differentiation

Hand differentiation involves computing derivatives of a numerical discretization by hand and writing a new code for computing the sensitivities. This approach has been used in some early works but it is very cumbersome and prone to errors. Moreover it has to be repeated whenever a change is made to the computer code. In the finite difference method, the cost function is evaluated by perturbing the control variable and then using a finite difference formula for approximating the derivatives. This is ideal for black-box situations where we do not have the source code of the flow solver and the number of design variables is small. But it is costly and prone to round-off errors [**?**] since the step size in the finite difference formula is very critical. The complex variable method [**?**, **?**] does not have the sensitivity to step size but is still costly since it requires complex arithmetic.

2 Automatic differentiation

Automatic differentiation (AD) is a technique for differentiating functions which are evaluated by a computer program. A computer program consists of elementary functions whose derivatives are known. In AD, the chain rule of differentiation is applied to a computer code to generate a new code for computing derivatives. There are several softwares available today which can perform automatic differentiation like ADIFOR, ADOLC,

TAMC, TAF, ODYSSEE, TAPENADE, etc. A useful place to find information on these tools is <http://www.autodiff.org>. AD can be performed in two modes known as *forward/direct* mode and *backward/reverse* mode, which will be explained shortly. In the present work we have used an AD tool called *Tapenade* [**?**] which is being developed by INRIA [**?**].

It is best to study AD using simple examples. Let us consider a function with two independent variables x, y

$$f = (xy + \sin x + 4)(3y^2 + 6)$$

In a computer implementation, a complicated function is broken down into several simpler expressions by introducing *intermediate* variables. The present example function can be evaluated as follows:

$$\begin{aligned}
t_1 &= x \\
t_2 &= y \\
t_3 &= t_1 t_2 \\
t_4 &= \sin t_1 \\
t_5 &= t_3 + t_4 \\
t_6 &= t_5 + 4 \\
t_7 &= t_2^2 \\
t_8 &= 3t_7 \\
t_9 &= t_8 + 6 \\
t_{10} &= t_6 t_9
\end{aligned}$$

where t_1, t_2 are the *independent* variables, $t_3, \ldots, t_9$ are the intermediate variables and $f = t_{10}$ is the *output* or *dependent* variable. In the present example all the intermediate variables are *active* since all of them affect the value of the output. Fig. (1) shows the implementation of the above function evaluation in Fortran 77.

2.1 Direct mode

In the direct mode, the standard chain rule of differential calculus is applied to the given function. To illustrate this let us define the following index set

$$I_k := \{i : i < k \text{ and } t_k \text{ depends explicitly on } t_i\}$$

Let α denote any of the independent variables (x and y in the present case) and let us define the partial derivatives

$$\dot{t}_i := \frac{\partial t_i}{\partial \alpha}, \quad t_{i,k} := \frac{\partial t_i}{\partial t_k}$$

Applying chain rule of differentiation to the k'th variable we have

$$\dot{t}_k = \frac{\partial t_k}{\partial \alpha} = \sum_{i \in I_k} \frac{\partial t_i}{\partial \alpha} \frac{\partial t_k}{\partial t_i} = \sum_{i \in I_k} \dot{t}_i t_{k,i} \quad k = 1, 2, \ldots, 9, 10$$

```
subroutine costfunc(x, y, f)
t1 = x
t2 = y
t3 = t1*t2
t4 = sin(t1)
t5 = t3 + t4
t6 = t5 + 4
t7 = t2**2
t8 = 3.0*t7
t9 = t8 + 6.0
t10 = t6*t9
f  = t10
end
```

Figure 1: Fortran 77 implementation

Note that the differentiation proceeds from the first to the last variable, i.e., in the same order as the function evaluation. For the example considered here, the chain rule gives the following result:

t_1	$=$	x	$\dot{t}_1$	$=$	$\dot{x}$
t_2	$=$	y	$\dot{t}_2$	$=$	$\dot{y}$
t_3	$=$	$t_1 t_2$	$\dot{t}_3$	$=$	$\dot{t}_1 t_2 + t_1 \dot{t}_2$
t_4	$=$	$\sin(t_1)$	$\dot{t}_4$	$=$	$\cos(t_1)\dot{t}_1$
t_5	$=$	$t_3 + t_4$	$\dot{t}_5$	$=$	$\dot{t}_3 + \dot{t}_4$
t_6	$=$	$t_5 + 4$	$\dot{t}_6$	$=$	$\dot{t}_5$
t_7	$=$	t_2^2	$\dot{t}_7$	$=$	$2t_2\dot{t}_2$
t_8	$=$	$3t_7$	$\dot{t}_8$	$=$	$3\dot{t}_7$
t_9	$=$	$t_8 + 6$	$\dot{t}_9$	$=$	$\dot{t}_8$
t_{10}	$=$	$t_6 t_9$	$\dot{t}_{10}$	$=$	$\dot{t}_6 t_9 + t_6 \dot{t}_9$

If we set $\dot{x} = 1$, $\dot{y} = 0$ then $\frac{\partial f}{\partial x} = \dot{t}_{10}$ while if $\dot{x} = 0$, $\dot{y} = 1$ then $\frac{\partial f}{\partial y} = \dot{t}_{10}$. In general, $\dot{t}_{10}$ gives the directional derivative along the direction $(\dot{x}, \dot{y})$. The Fortran subroutine can be differentiated using the following command:

```
tapenade -forward -vars "x y" -outvars "f" costfunc.f
```

where `costfunc.f` is the name of the file containing the subroutine. The output is a new Fortran file named `costfunc_d.f` containing the subroutine as shown in fig. (2). Note that corresponding to each active variable a new variable has been introduced by appending a `d` at the end. Thus we have `xd` corresponding to `x`, `yd` corresponding to `y`, etc., and in terms of the previous mathematical notation, `xd`=$\dot{x}$, `yd`=$\dot{y}$, etc. Comparing the output of AD to the above mathematical derivation, we see that AD is also applying the chain rule of differentiation. Each line is differentiated and is added above the original line. If we want to compute both the partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, we have to call the differentiated subroutine twice with appropriate values initialized for `xd` and `yd`.

```
SUBROUTINE COSTFUNC_D(x, xd, y, yd, f, fd)
t1d = xd
t1 = x
t2d = yd
t2 = y
t3d = t1d*t2 + t1*t2d
t3 = t1*t2
t4d = t1d*COS(t1)
t4 = SIN(t1)
t5d = t3d + t4d
t5 = t3 + t4
t6d = t5d
t6 = t5 + 4
t7d = 2*t2*t2d
t7 = t2**2
t8d = 3.0*t7d
t8 = 3.0*t7
t9d = t8d
t9 = t8 + 6.0
t10d = t6d*t9 + t6*t9d
t10 = t6*t9
fd = t10d
f = t10
END
```

Figure 2: Automatic differentiation in direct mode

2.2 Reverse mode

In the direct mode, the chain rule of differentiation was applied from the first variable to the last. But it can also be applied in the reverse order. First define the following index set,

$$J_k := \{i : i > k \text{ and } t_i \text{ depends explicitly on } t_k\}$$

and let

$$\bar{t}_i := \frac{\partial f}{\partial t_i} = \frac{\partial t_{10}}{\partial t_i}, \quad t_{i,k} := \frac{\partial t_i}{\partial t_k}$$

We can apply chain rule of differentiation in reverse as follows,

$$\bar{t}_k = \frac{\partial t_{10}}{\partial t_k} = \sum_{i \in J_k} \frac{\partial t_{10}}{\partial t_i} \frac{\partial t_i}{\partial t_k} = \sum_{i \in J_k} \bar{t}_i t_{i,k} \quad k = 10, 9, \ldots, 2, 1$$

Applying this reverse differentiation procedure to the example function, we obtain

t_1	=	x	$\bar{t}_{10}$	=	1	
t_2	=	y	$\bar{t}_9$	=	$\bar{t}_{10} t_{10,9}$	= t_6
t_3	=	$t_1 t_2$	$\bar{t}_8$	=	$\bar{t}_9 t_{9,8}$	= t_6
t_4	=	$\sin(t_1)$	$\bar{t}_7$	=	$\bar{t}_8 t_{8,7}$	= $3t_6$
t_5	=	$t_3 + t_4$	$\bar{t}_6$	=	$\bar{t}_{10} t_{10,6}$	= t_9
t_6	=	$t_5 + 4$	$\bar{t}_5$	=	$\bar{t}_6 t_{6,5}$	= t_9
t_7	=	t_2^2	$\bar{t}_4$	=	$\bar{t}_5 t_{5,4}$	= t_9
t_8	=	$3t_7$	$\bar{t}_3$	=	$\bar{t}_5 t_{5,3}$	= t_9
t_9	=	$t_8 + 6$	$\bar{t}_2$	=	$\bar{t}_7 t_{7,2} + \bar{t}_3 t_{3,2}$	= $6t_2 t_6 + t_1 t_9$
t_{10}	=	$t_6 t_9$	$\bar{t}_1$	=	$\bar{t}_4 t_{4,1} + \bar{t}_3 t_{3,1}$	= $t_9 \cos(t_1) + t_9 t_2$

Note that unlike the direct mode, the function evaluation and its differentiation do not go together. The function must be evaluated first and all the variables must be initialized. Then reverse differentiation is executed. The partial derivatives are given by $\frac{\partial f}{\partial x} = \bar{t}_1$ and $\frac{\partial f}{\partial y} = \bar{t}_2$. All the required partial derivatives are obtained in a single computation. The subroutine can be differentiated with Tapenade as follows

```
tapenade -backward -vars "x y" -outvars "f" costfunc.f
```

and the output subroutine shown in fig. (3) is contained in a file named `costfunc_b.f` where the `_b` denotes backward differentiation. We see that the initial lines of the program evaluate the intermediate variables which are required for reverse mode differentiation; then the reverse mode differentiation is executed. The variables `t1b, t2b, ...` etc. correspond to $\bar{t}_1, \bar{t}_2, \ldots$ etc.

2.3 Comparison of direct and reverse mode

For a scalar function f, the direct mode gives the tangential derivative $\nabla f \cdot \vec{s}$ for some given vector $\vec{s}$, while the reverse mode gives all the components of ∇f. For a vector function

```
SUBROUTINE COSTFUNC_B(x, xb, y, yb, f, fb)
t1 = x
t2 = y
t3 = t1*t2
t4 = SIN(t1)
t5 = t3 + t4
t6 = t5 + 4
t7 = t2**2
t8 = 3.0*t7
t9 = t8 + 6.0
t10b = fb
t6b = t9*t10b
t9b = t6*t10b
t8b = t9b
t7b = 3.0*t8b
t5b = t6b
t3b = t5b
t2b = t1*t3b + 2*t2*t7b
t4b = t5b
t1b = t2*t3b + COS(t1)*t4b
yb = t2b
xb = t1b
fb = 0.0
END
```

Figure 3: Automatic differentiation in reverse mode

R, the direct mode AD gives $\nabla R \cdot \vec{s}$ while reverse mode gives $(\nabla R)^{\mathrm{T}} \cdot \vec{s}$. If we want to know all the components of the jacobian matrix then the above jacobian products have to be computed for different values of unit tangent vector. In most practical applications the full jacobian ∇R is not required and it is enough to be able to compute its product with some specified vector.

In reverse mode AD, intermediate variables which affect the output are required in reverse order. Hence these must be computed and stored before the reverse differentiation can start. In a computer code, it is quite common to use the same memory location to store different values at different points of the program. If these values are required in the reverse mode, then Tapenade will store them in a stack using a PUSH function [**?**]. These values are then recalled from the stack by using a POP function during the reverse differentiation. Calls to PUSH/POP functions must be minimized as much as possible since they can slow down the execution of the program.

Most numerical computations including those in CFD make use of iterative methods. Only the final converged solution is of interest. But an AD tool like Tapenade cannot distinguish such situations and it will differentiate the whole iterative sequence. This leads to enormous memory requirements since all the intermediate solutions will be stored in stack. For most applications, these huge memory requirements will rule out the reverse mode [**?**]. In order to overcome the memory barrier, reverse mode AD must be applied in a *piecemeal* manner [**?**]. The iterative solver must be written with a highly modular structure using subroutines and functions, and AD is then used to differentiate these individual modules. The differentiated modules are used to construct an iterative solver for the adjoint equations. To illustrate this approach, it is instructive to first look at the nature of the adjoint equations at the level of each cell or grid point.

2.4 Verification of adjoint code

Suppose we have developed the adjoint code for computing

$$\left(\frac{\partial R}{\partial u}\right)^{\mathrm{T}} v = A^{\mathrm{T}} v, \qquad A(u) := \frac{\partial}{\partial u} R(u) \tag{2}$$

i.e. given the vector v the adjoint code returns the matrix-vector product $A^{\mathrm{T}}v$. But we have the following identity

$$x^{\mathrm{T}} A y = y^{\mathrm{T}} A^{\mathrm{T}} x \tag{3}$$

for any two vectors x and y. An approximation to Ay can be obtained using finite differences

$$Ay \approx \frac{1}{\epsilon}[R(u + \epsilon y) - R(u)] \tag{4}$$

for some small $\epsilon > 0$. The correctness of the AD generated adjoint code can be verified using the dot-product test:

- Choose a random vector y whose elements lie in $(-1, +1)$ and a small $\epsilon > 0$

- Compute

$$t = \frac{1}{\epsilon}[R(u + \epsilon y) - R(u)] \tag{5}$$

- Set $x = 1$ and compute $s = A^{\mathrm{T}}x$ using adjoint code.
- Verify that $x^{\mathrm{T}}t \approx y^{\mathrm{T}}s$

Since t is a finite difference approximation, we have to try out different values of ϵ to get a good agreement between the two values in the last step.

3 Implicit solution of adjoint equation

The adjoint equation is

$$A^{\mathrm{T}}V = b, \quad A = \frac{\partial R}{\partial U}, \quad b = -\frac{\partial J}{\partial U} \tag{6}$$

This equation can be solved using a pseudo time integration scheme

$$\frac{\partial V}{\partial t} + A^{\mathrm{T}}V = b, \quad V = 0 \text{ at } t = 0 \tag{7}$$

An implicit scheme can be written for this equation as

$$\frac{\Delta V^n}{\Delta t} + A^{\mathrm{T}}V^{n+1} = b \text{ or} \tag{8}$$

or

$$\left[\frac{I}{\Delta t} + A^{\mathrm{T}}\right]\Delta V^n = -A^{\mathrm{T}}V^n + b \tag{9}$$

Since we are only interested in the steady solution the matrix A^{T} on the left hand side can be any stable approximation. An efficient approximation is the diagonal form proposed by Pulliam and Chaussee. For the 2-D Euler equations $U_t + R(U) = 0$ this takes the form

$$T_\xi[I + h\delta_\xi\Lambda_\xi]T_\xi^{-1}T_\eta[I + h\delta_\eta\Lambda_\eta]T_\eta^{-1}\Delta U^n = -hR(U^n) \tag{10}$$

In order to derive an implicit scheme for the adjoint let us start from the discrete finite volume residual

$$R_{i,j} = F_{i+1/2,j} - F_{i-1/2,j} + F_{i,j+1/2} - F_{i,j-1/2} \tag{11}$$

where $F_{i+1/2,j} = F(U_{i,j}, U_{i+1,j}, \hat{s}_{i+1/2,j})$ is a numerical flux function. The first order adjoint equation for the cell (i, j) is given by

$$\begin{aligned} &-\mathcal{A}^+_{i+1/2,j}(V_{i+1,j} - V_{i,j}) - \mathcal{A}^-_{i-1/2,j}(V_{i,j} - V_{i-1,j}) \\ &-\mathcal{A}^+_{i,j+1/2}(V_{i,j+1} - V_{i,j}) - \mathcal{A}^-_{i,j-1/2}(V_{i,j} - V_{i,j-1}) \quad = b_{i,j} \end{aligned}$$

where $\mathcal{A}^+ = [\partial_X F(X, Y, \hat{s})]^{\mathrm{T}}$ and $\mathcal{A}^- = [\partial_Y F(X, Y, \hat{s})]^{\mathrm{T}}$. Using the first order adjoint we can write an implicit equation as

$$[I - \mathcal{A}^+_{i+1/2,j}\delta^+_\xi - \mathcal{A}^-_{i-1/2,j}\delta^-_\xi - \mathcal{A}^+_{i,j+1/2}\delta^+_\eta - \mathcal{A}^-_{i,j-1/2}\delta^-_\eta]\Delta V_{i,j} = -R^*_{i,j} \tag{12}$$

where the time step and cell volume are not explicitly indicated and the residual on the right is the second order adjoint residual. This can be approximately factorized as

$$[I - \mathcal{A}^+_{i+1/2,j}\delta^+_\xi - \mathcal{A}^-_{i-1/2,j}\delta^-_\xi][I - \mathcal{A}^+_{i,j+1/2}\delta^+_\eta - \mathcal{A}^-_{i,j-1/2}\delta^-_\eta]\Delta V_{i,j} = -R^*_{i,j} \tag{13}$$

Using a diagonal factorization scheme of Pulliam and Chaussee we can write the implicit scheme as

$$T_\xi^{-\mathrm{T}}[I - \Lambda^+_\xi\delta^+_\xi - \Lambda^-_\xi\delta^-_\xi]T^{\mathrm{T}}_\xi T_\eta^{-\mathrm{T}}[I - \Lambda^+_\eta\delta^+_\eta - \Lambda^-_\eta\delta^-_\eta]\Delta V = -R^* \tag{14}$$