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.

Adjoint code development and optimization using automatic differentiation
Computational and Theoretical Fluid Dynamics

Author: Praveen. C

Published by National Aerospace Laboratories, India.

Typesetting: Pages created by author using LaTeX macro package.

Online versions of this document are available at:
http://nal-ir.nal.res.in

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 $\delta\alpha = -\epsilon G^\top$ for some $\epsilon > 0$, since $\delta J = -\epsilon G^\top G = -\epsilon \|G\|^2 \leq 0$. 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

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

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

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

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

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

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 [11] since the step size in the finite difference formula is very critical. The complex variable method [2, 9] 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* [6] which is being developed by INRIA [1].

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.

```
      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

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

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

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`.

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

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

```
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

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 R, the direct mode AD gives $\nabla R \cdot \vec{s}$ while reverse mode gives $(\nabla R)^\top \cdot \vec{s}$. 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 [7]. 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

```
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

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 [12]. In order to overcome the memory barrier, reverse mode AD must be applied in a *piecemeal* manner [5]. 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.

3. Discrete one-D adjoint equations

Consider a one dimensional system governed by a conservation law. A finite volume discretization at steady state leads to

$$R_i := F_{i+1/2} - F_{i-1/2} = 0$$

where R_i is the finite volume residual for i'th cell. The interfacial fluxes are computed by using a numerical flux function

$$F = F(X, Y), \quad F_{i+1/2} = F(U_i, U_{i+1})$$

If the control variable α is perturbed by an amount $\delta\alpha$, then the perturbation in the finite volume residual is

$$\begin{aligned} \delta R_i = \quad & \tfrac{\partial}{\partial X} F_{i+1/2} \delta U_i + \tfrac{\partial}{\partial Y} F_{i+1/2} \delta U_{i+1} \\ & - \tfrac{\partial}{\partial X} F_{i-1/2} \delta U_{i-1} - \tfrac{\partial}{\partial Y} F_{i-1/2} \delta U_i + \tfrac{\partial R_i}{\partial \alpha} \delta\alpha \quad = 0 \end{aligned}$$

Introduce adjoint variable V_i for i'th cell

$$\delta J = \frac{\partial J}{\partial \alpha} \delta\alpha + \sum_i \frac{\partial J}{\partial U_i} \delta U_i + \sum_i V_i^\top \delta R_i$$

and collecting terms containing δU_i

$$\begin{aligned} \delta J = \sum_i \Bigg[\frac{\partial J}{\partial U_i} \quad & + \quad V_{i-1}^\top \frac{\partial}{\partial Y} F_{i-1/2} \\ & + \quad V_i^\top \left(\frac{\partial}{\partial X} F_{i+1/2} - \frac{\partial}{\partial Y} F_{i-1/2} \right) \\ & - \quad V_{i+1}^\top \frac{\partial}{\partial X} F_{i+1/2} \Bigg] \delta U_i \; + \; \ldots \end{aligned}$$

The adjoint equation for i'th cell is obtained by setting the coefficient of δU_i to zero

$$\begin{aligned} R_i^* := \left(\frac{\partial J}{\partial U_i} \right)^\top + \left(\frac{\partial}{\partial Y} F_{i-1/2} \right)^\top V_{i-1} \quad & + \quad \left(\frac{\partial}{\partial X} F_{i+1/2} - \frac{\partial}{\partial Y} F_{i-1/2} \right)^\top V_i \\ & - \quad \left(\frac{\partial}{\partial X} F_{i+1/2} \right)^\top V_{i+1} = 0 \end{aligned}$$

```
While u is not converged
  res = 0.0
  fluxinflow(u(1), res(1))
  do i=1,N-1
    fluxinterior(u(i), u(i+1), res(i), res(i+1))
  enddo
  fluxoutflow(u(N), res(N))
  do i=1,N
    u(i) = u(i) - (dt/dx)*res(i)
  enddo
endwhile

cost=0.0
costfunc(u, cost)
```

Figure 4: Iterative solver for primal problem $R = 0$

We note that the adjoint equation is linear in V and the coefficients depend on the solution U of the primal problem. The terms containing the adjoint variable are of the form of a product between the transpose of the flux jacobian and the adjoint variable. This suggests the use of reverse mode AD to differentiate the flux subroutines. The adjoint equation $R^* = 0$ is solved iteratively by introducing a pseudo-time term

$$\Delta x \frac{\mathrm{d}V_i}{\mathrm{d}t} + R_i^* = 0$$

The pseudo-code listed in fig. (4) uses explicit Euler time-stepping scheme to solve the primal problem. The variable `res` contains the finite volume residual; the flux subroutines compute the interfacial flux and add it to the finite volume residual. For an interior cell face, the flux across the face is added to the residual of the left cell and subtracted from the residual of the right cell within the flux subroutine.

The iterative solver for the adjoint variable is listed in fig. (5). All the subroutines are differentiated with respect to the state `u` in reverse mode. For example the interior flux subroutine which is defined as (this is just an example)

```
subroutine fluxinterior(ul, ur, resl, resr)
real ul, ur, resl, resr, flux
flux = (ul**2 + ur**2)/2
resl = resl + flux
resr = resr - flux
return
end
```

is differentiated as

```
costb = 1.0
ub1   = 0.0
costfunc_b(u, ub1, cost, costb)

v   = 0.0
res = 0.0
While v is not converged
  ub2 = 0.0
  fluxinflow_b(u(1), ub2(1), res(1), v(1))
  do i=1,N-1
    fluxinterior_b(u(i), ub2(i), u(i+1), ub2(i+1),
                   res(i), v(i), res(i+1), v(i+1))
  enddo
  fluxoutflow_b(u(N), ub2(N), res(N), v(N))
  do i=1,N
    v(i) = v(i) - (dt/dx)*(ub1(i) + ub2(i))
  enddo
endwhile
```

Figure 5: Iterative solver for adjoint equation

```
tapenade -backward -head fluxinterior -vars "ul ur resl resr" \\
         -outvars "ul ur resl resr" routines.f
```

The variable `ub1` contains $\frac{\partial J}{\partial u}$; this is computed once and stored since it does not depend on the adjoint variable. The variable `ub2` contains $(\frac{\partial R}{\partial u})^\top V$; the adjoint residual is given by $R^* =$ `ub1 + ub2` which is used in a time-stepping scheme to update the adjoint variable. Note that there is no need to reverse the order of flux computations since the contribution of each flux evaluation is independent of others, and merely gets added to the variable `ub2`.

4. Quasi 1-D flow in a duct

Consider flow in a duct where the flow variations across the duct cross-section are ignored. The governing equations in conservation form are

$$\frac{\partial}{\partial t}(hU) + \frac{\partial}{\partial x}(hf) = \frac{\mathrm{d}h}{\mathrm{d}x}P, \quad x \in (a,b) \quad t > 0$$

where $h(x)$ is the cross section height of the duct, U is the vector of conserved variables and f is the flux vector.

$$U = \begin{bmatrix} \rho \\ \rho u \\ E \end{bmatrix}, \quad f = \begin{bmatrix} \rho u \\ p + \rho u^2 \\ (E+p)u \end{bmatrix}, \quad P = \begin{bmatrix} 0 \\ p \\ 0 \end{bmatrix}$$

An inverse design problem for this system is: find the shape of the duct h which gives pressure distribution p^*. This can be put as an optimization problem: find the shape h which minimizes

$$J = \int_a^b (p - p^*)^2 \mathrm{d}x$$

The governing equations are solved using a finite volume scheme

$$h_i \frac{\mathrm{d}U_i}{\mathrm{d}t} + \frac{h_{i+1/2}F_{i+1/2} - h_{i-1/2}F_{i-1/2}}{\Delta x} = \frac{(h_{i+1/2} - h_{i-1/2})}{\Delta x} P_i$$

and the finite volume residual is

$$R_i := (h_{i+1/2}F_{i+1/2} - h_{i-1/2}F_{i-1/2}) - (h_{i+1/2} - h_{i-1/2})P_i$$

The cost function is discretized as

$$J = \sum_{i=1}^{N} (p_i - p_i^*)^2$$

In the discrete problem, the design variables are the values of the duct height at the cell face locations

$$\alpha = \{h_{1/2}, h_{1+1/2}, \ldots, h_{i+1/2}, \ldots, h_{N+1/2}\}$$

where N is the number of cells.

The construction of the adjoint iterative solver follows the steps given in the previous sections. An additional computation arises due to the presence of the source term which is implemented in a separate subroutine. The flow and adjoint equations are marched in time to steady state using explicit Euler time stepping.

4.1. Adjoint solution

The shape corresponding to the target pressure p^* is

$$h(x) = a + b \tanh(cx - d)$$

where a, b are determined from the specified inlet and outlet heights (1.0512 and 1.75), while $c = c^* = 0.8$ and $d = d^* = 4$. We start with the shape corresponding to $c = 1.0$ and $d = 3.8$ with the inlet and exit heights being held fixed. Note that the above formula is used only to define the target and initial shapes. The shape modification during optimization is achieved by modifying the discrete set of control variables α indicated in the previous section. The target and starting shapes are shown in fig. (6) while the corresponding pressure distributions are shown in fig. (7). The flow solution is obtained using three different flux functions: AUSMDV [15], KFVS [10] and Lax-Friedrichs [16]. These flux functions have varying amounts of numerical dissipation with the Lax-Friedrichs flux being the most dissipative, as is evident from the width of the shock region in fig. (7).

The adjoint variable corresponding to the density equation is shown in fig. (8) along with the convergence history. It is seen that the adjoint solution is continuous even where the shock is present. This is true of the other two components of the adjoint solution also. The AUSMDV and KFVS fluxes give nearly the same adjoint solution while the Lax-Friedrichs flux shows greater discrepancy in the pre-shock region. The convergence plot shows that both the flow and

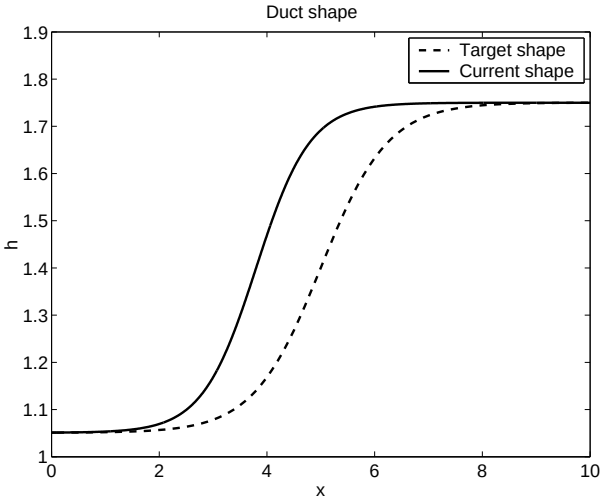


Figure 6: Target and starting shape

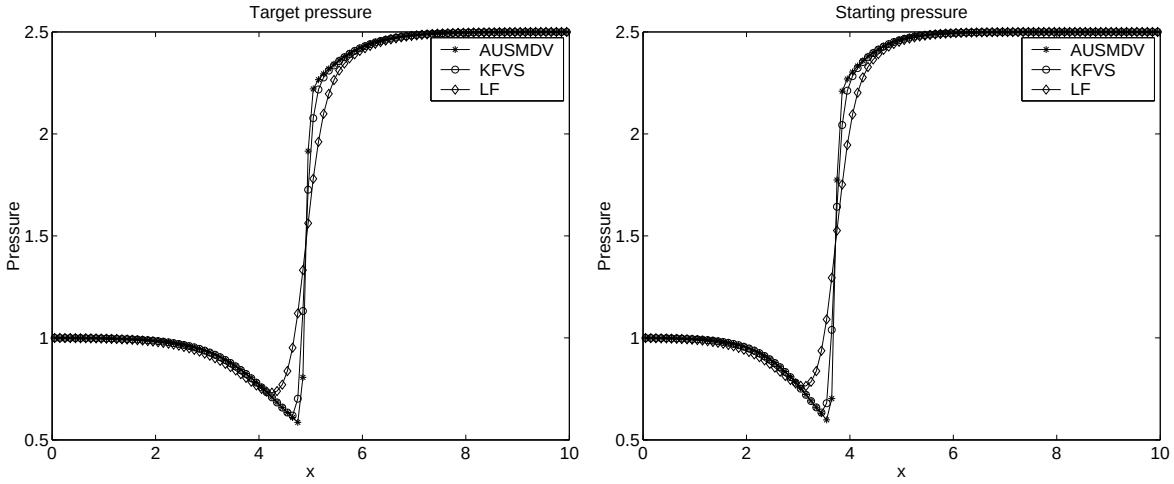


Figure 7: Target and starting pressure distribution

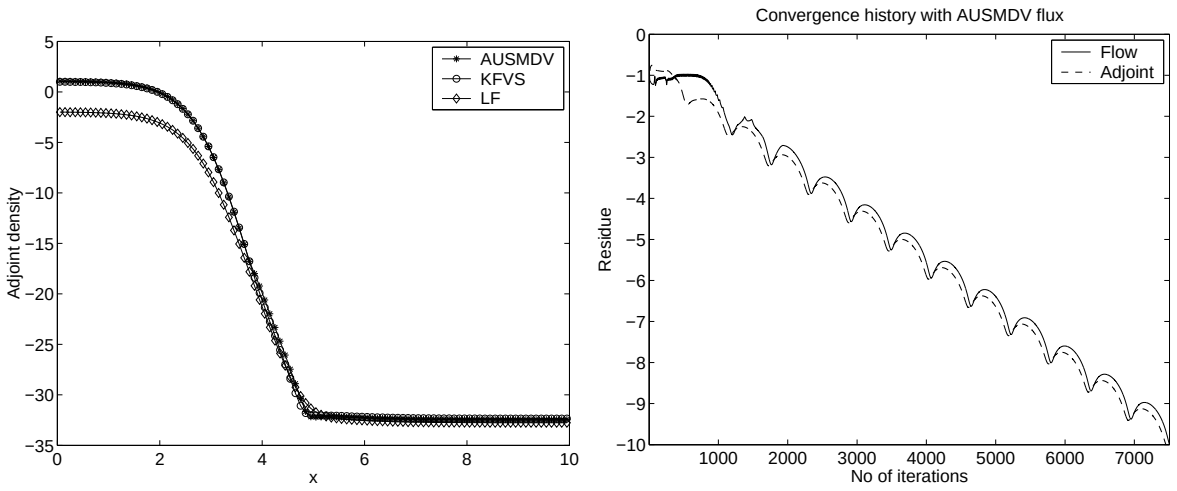


Figure 8: Adjoint density and convergence history

adjoint solutions converge at nearly the same rate which is to be expected since both equations have the same eigenvalues and time-step restriction.

The gradient obtained from the adjoint solution is shown in fig. (9). In smooth regions, both AUSMDV and KFVS fluxes yield very similar gradients while Lax-Friedrichs flux shows considerable difference from these two. In the shock region, the peak value of the gradients is different for all the fluxes, with AUSMDV showing an abrupt spike while Lax-Friedrich flux gives a smoother variation.

4.2. Validation of gradients

The adjoint solution can be indirectly validated by checking the accuracy of the gradients. The gradients obtained using adjoint solution are compared with finite difference (FD) approximations. However a finite difference approximation will not be exact; it is contaminated by the discretization error and round-off error. The error depends on the step size used in a finite difference formula; a large step size leads to errors which are dominated by discretization error while a very small step size leads to errors which are dominated by roundoff. In fact there will be an optimum step size at which the error in the finite difference approximation is least. This optimum size cannot be determined in practice since it depends on many factors like the smoothness and complexity of the function involved, the finite difference formula used, the accuracy of the computer, etc.

For the quasi 1-D flow, the shape h is perturbed by a small amount Δh and the cost function is re-evaluated. The gradient is obtained using a central difference formula

$$\frac{\partial J}{\partial h} \approx \frac{J(h + \Delta h) - J(h - \Delta h)}{2\Delta h}$$

Since there are 101 control variables, we choose only three of them for validation at locations shown in fig. (10). The FD gradients are computed with varying step sizes Δh and are listed in table (1) along with the value obtained from AD. The AD solution should be exact to machine precision and we expect it to agree with the FD solution at some intermediate value of step

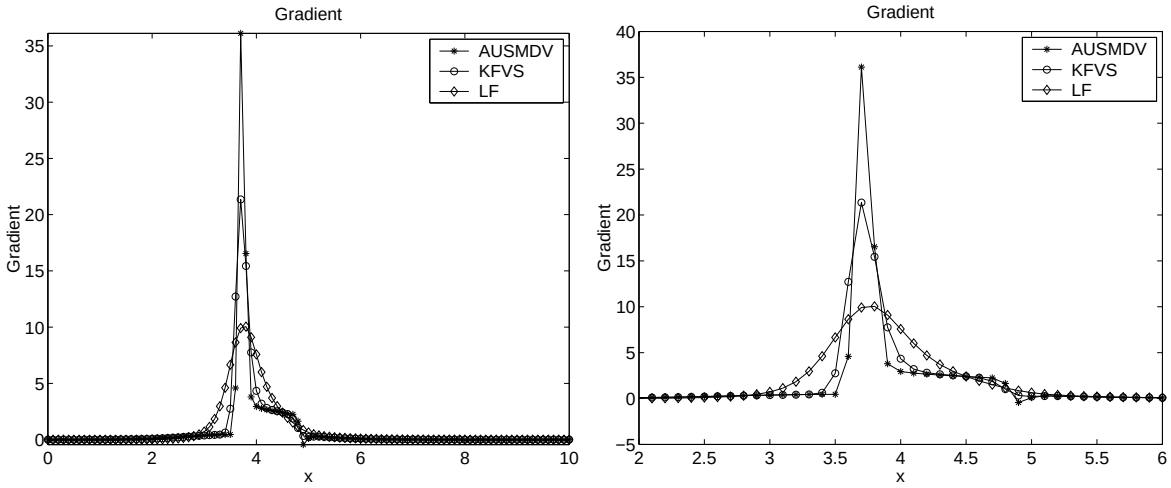


Figure 9: Shape gradient

Δh	A	B	C
0.01	0.4191069499	35.18452823	2.545316345
0.001	0.4231223499	36.10982621	2.556461900
0.0001	0.4231624999	36.11933154	2.556573499
0.00001	0.4231599998	36.11942125	2.556575000
0.000001	0.4231999994	36.11942305	2.556550001
0.0000001	0.4229999817	36.11942329	2.556499971
AD	0.4231628330	36.11941951	2.556574450

Table 1: Validation of shape gradient for quasi 1-D flow

size. This behaviour is indeed observed in the table since the gradients computed from AD lie between those of FD corresponding to $\Delta h = 0.0001$ and $\Delta h = 0.00001$. Location B is inside the shock and we note that AD is still giving correct gradients.

4.3. Gradient smoothing

The gradients computed from the discrete adjoint equations may not be smooth due to non-differentiable functions and presence of shocks. If the control variables are updated using non-smooth gradients then it may lead to physically unrealistic solutions. In shape optimization, the new shape may have sharp turns and corners which will spoil the flow. Hence it is necessary to smooth the gradients before using them in a minimization algorithm. The gradients are smoothed using an elliptic equation

$$\left(1 - \epsilon_i \frac{\mathrm{d}^2}{\mathrm{d}x^2}\right) \bar{g}_i = g_i$$

where g is the unsmoothed gradient, $\bar{g}$ is the smoothed gradient and the parameter ϵ controls the amount of smoothing. In regions where the gradient is smooth, ϵ must be small and where it

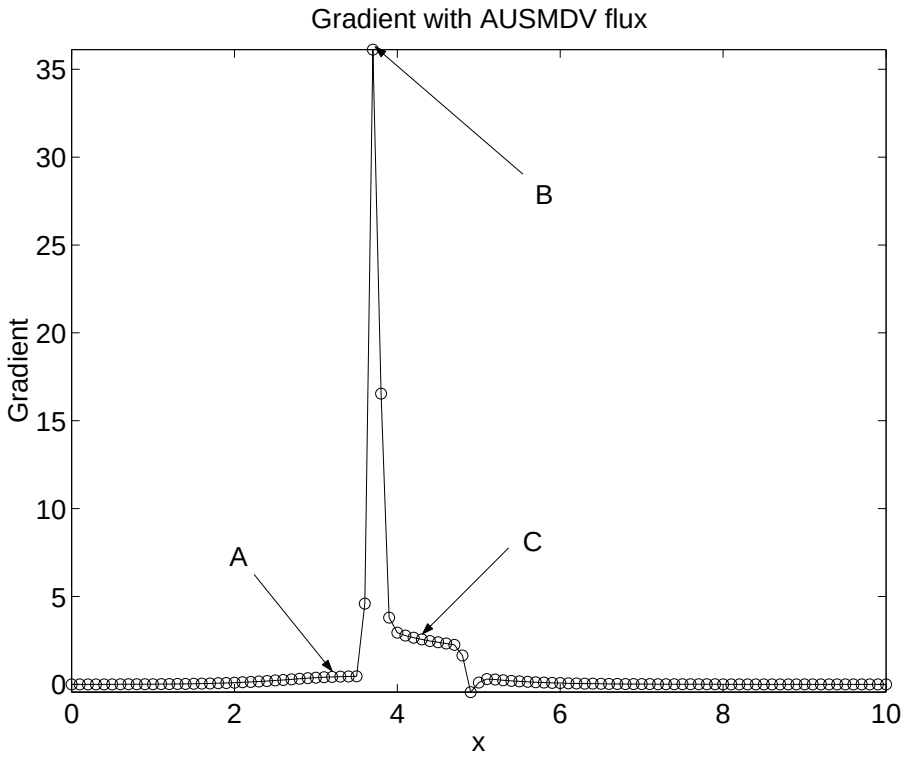


Figure 10: Validations of gradients at specific points A, B and C

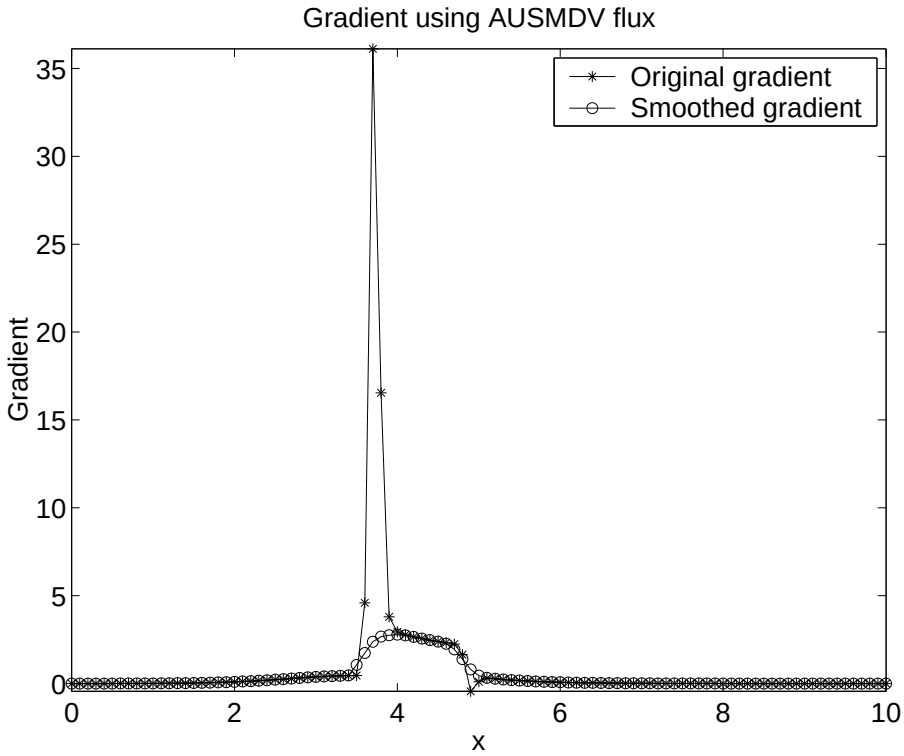


Figure 11: Unsmoothed and smoothed gradients for pressure matching problem

is not sufficiently smooth, ϵ is chosen to be large. Since ϵ should depend on the local smoothness we borrow ideas from limiters and choose

$$\epsilon_i = \{|g_{i+1} - g_i| + |g_i - g_{i-1}|\}L_i$$

$$L_i = \frac{|g_{i+1} - 2g_i + g_{i-1}|}{\max(|g_{i+1} - g_i| + |g_i - g_{i-1}|, \mathrm{TOL})}$$

The elliptic equation is solved using a finite difference scheme with Jacobi iterations. The finite difference scheme is given by

$$\bar{g}_i - \frac{2\epsilon_i}{x_{i+1} - x_{i-1}} \left(\frac{\bar{g}_{i+1} - \bar{g}_i}{x_{i+1} - x_i} - \frac{\bar{g}_i - \bar{g}_{i-1}}{x_i - x_{i-1}} \right) = g_i$$

Fig. (11) shows an example of the elliptic smoothing applied to the gradients obtained for a pressure matching problem in a quasi 1-D flow through a duct. A large spike is observed in the raw gradients due to the presence of a shock which is smoothed out after applying the elliptic equation. Note that the smooth portions of the gradient are undisturbed by the elliptic smoother.

4.4. Optimization example

The adjoint solver and smoothing procedure are applied to solve a quasi 1-D pressure matching problem for flow through a duct. The minimization algorithm is a steepest descent method with

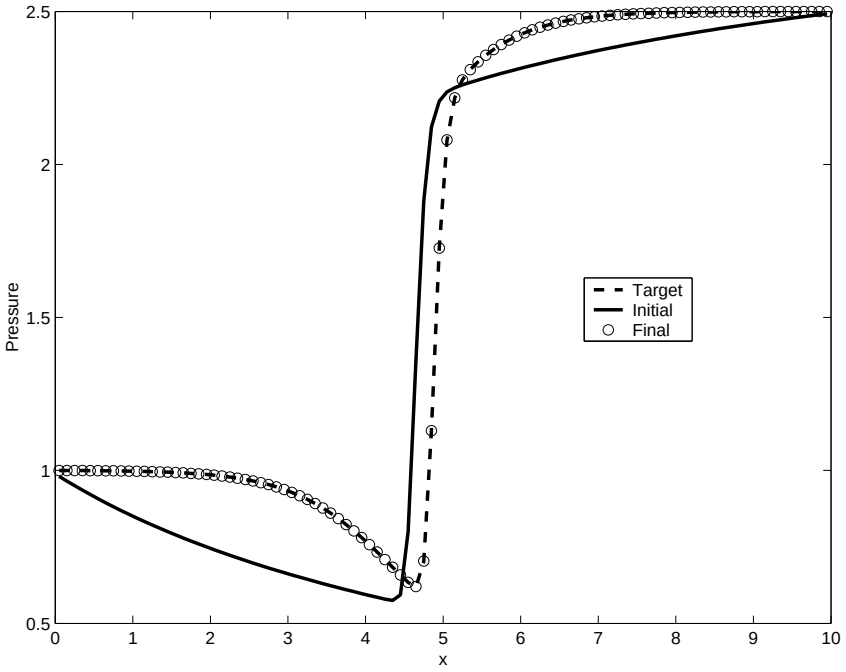


Figure 12: Target pressure, starting pressure and optimized pressure distribution

variable step size given by

$$\alpha_i^{n+1} = \alpha_i^n - S\left[\Delta\alpha_i^n \frac{\mathrm{d}J}{\mathrm{d}\alpha_i}(\alpha^n)\right]$$

where the step size for each control variable is increased by 50% if the gradient does not change sign and is reduced by 50% if it does.

$$\Delta\alpha_i^n = \begin{cases} \frac{3}{2}\Delta\alpha_i^{n-1} & \text{if } \frac{\mathrm{d}J}{\mathrm{d}\alpha_i}(\alpha^n)\frac{\mathrm{d}J}{\mathrm{d}\alpha_i}(\alpha^{n-1}) > 0 \\ \frac{1}{2}\Delta\alpha_i^{n-1} & \text{if } \frac{\mathrm{d}J}{\mathrm{d}\alpha_i}(\alpha^n)\frac{\mathrm{d}J}{\mathrm{d}\alpha_i}(\alpha^{n-1}) < 0 \end{cases}$$

The operator $S[\cdot]$ denotes the smoothing step using the elliptic smoother.

The target pressure is taken for the shape mentioned in section (4.1) but the starting shape is taken to be a straight diverging duct. The target, starting and optimized pressure distributions are shown in fig. (12) with the optimized pressure distribution obtained after 50 iterations. The shape obtained at the end of 50 iterations is shown in fig. (13). Both the pressure distribution and shape agree quite well with the target values. The convergence of the cost function and gradient is shown in fig. (14) with the cost function decreasing by five orders of magnitude and gradient by four orders of magnitude. In this problem the cost function does not explicitly depend on the control variable; hence as the optimum is approached the adjoint variable must converge to zero in order to satisfy the first order extremum condition. This is indeed observed as seen in fig. (15) where the adjoint solution after 50 steps is close to zero.

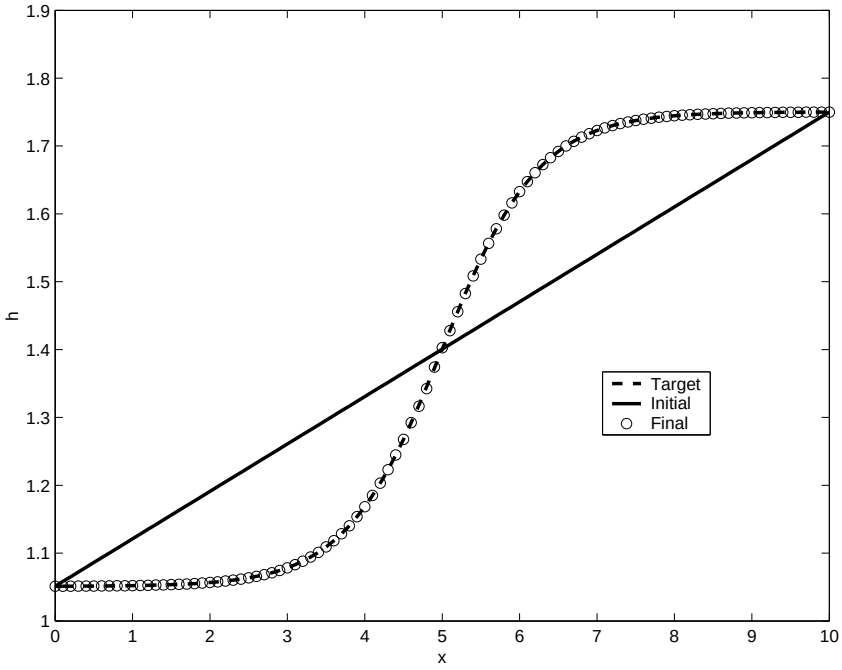


Figure 13: Target shape, initial shape and final optimized shape for pressure matching problem

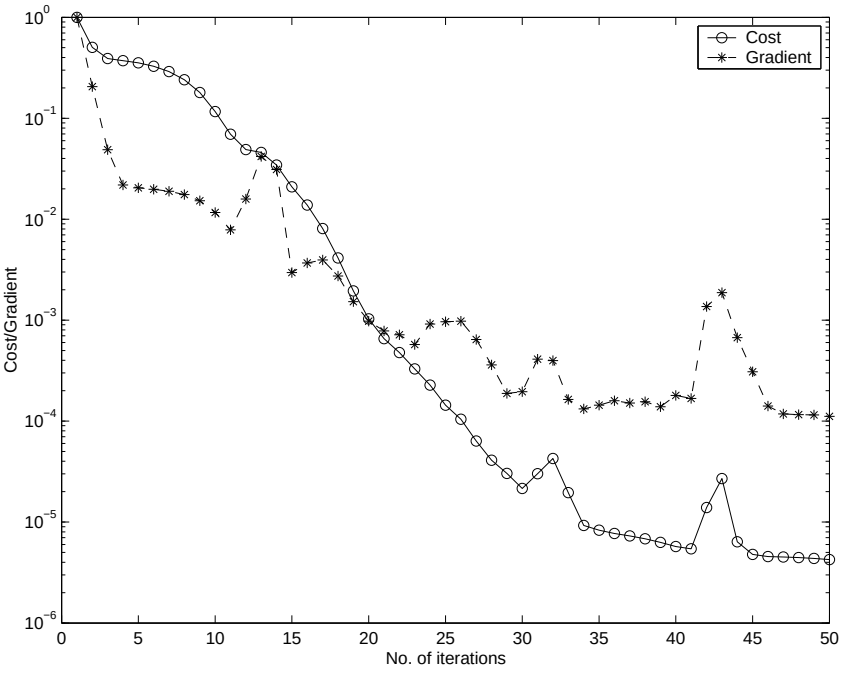


Figure 14: Convergence of cost function and gradient for pressure matching problem

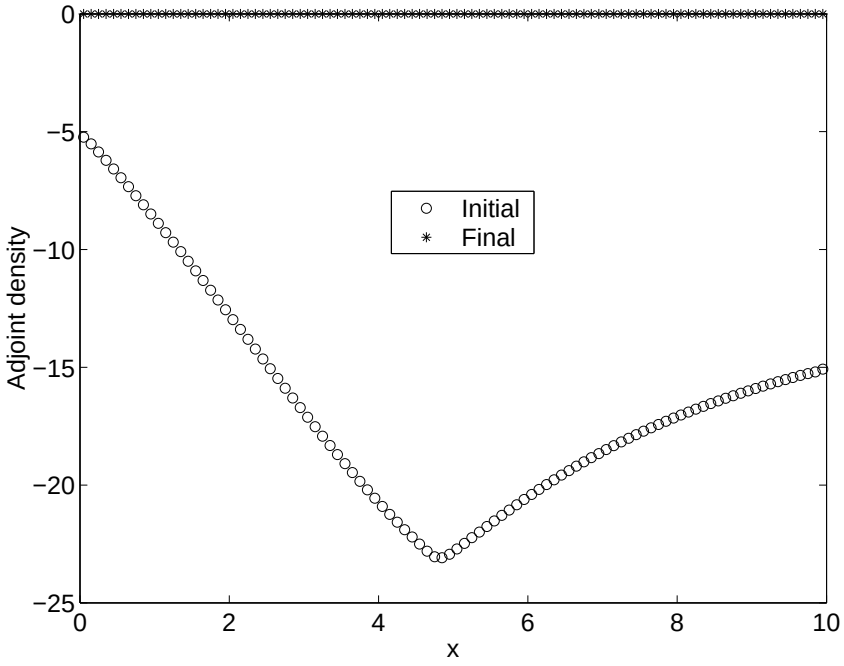


Figure 15: Initial and final adjoint solution corresponding to density equation for pressure matching problem

5. 2-D inviscid compressible flow

The 2-D Euler equations in conservation form are written as

$$\frac{\partial U}{\partial t} + \frac{\partial f}{\partial x} + \frac{\partial g}{\partial y} = 0$$

with

$$U = \begin{bmatrix} \rho \\ \rho u_x \\ \rho u_y \\ E \end{bmatrix}, \quad f = \begin{bmatrix} \rho u_x \\ p + \rho u_x^2 \\ \rho u_x u_y \\ (E+p)u_x \end{bmatrix}, \quad g = \begin{bmatrix} \rho u_y \\ \rho u_x u_y \\ p + \rho u_y^2 \\ (E+p)u_y \end{bmatrix}$$

where ρ is the fluid density, (u_x, u_y) are the components of fluid velocity, p is the static pressure and E is the total energy per unit volume given by

$$E = \frac{p}{\gamma - 1} + \frac{1}{2}\rho(u_x^2 + u_y^2)$$

The integral form of the conservation law for a control volume T is

$$\frac{\mathrm{d}}{\mathrm{d}t}\int_T U \mathrm{d}x\mathrm{d}y + \oint_{\partial T} (f n_x + g n_y)\mathrm{d}S = 0$$

5.1. 2-D finite volume solver

A vertex-centroid finite volume scheme similar to that of Frink [4], Jameson et al. [8] is used. The grid consists of triangular cells and the unknowns are stored at the triangle centroids. The

semi-discrete update equation for the i'th triangle T_i is

$$A_i \frac{\mathrm{d}U_i}{\mathrm{d}t} + \sum_{j \in N(i)} F_{ij} \Delta S_{ij} = 0, \quad U_i = \frac{1}{A_i} \int_{T_i} U(x,y,t) \mathrm{d}x \mathrm{d}y$$

where U_i is the cell-average value, A_i is the area of triangle T_i, $N(i)$ is the set of triangles adjacent to the i'th triangle, F_{ij} provides an approximation of the normal flux across the edge between T_i and T_j and ΔS_{ij} is the length of this edge. The interfacial flux is evaluated by reconstructing the states on either side of an edge and using a numerical flux function. The numerical flux function $F = F(X, Y, \hat{n})$ may be obtained from a flux vector or flux difference splitting scheme; the interface flux is given by $F(U_L, U_R, \hat{n})$. The reconstruction is performed using values at the cell centroids and vertices. Referring to fig. (16) the reconstructed states U_L, U_R are computed as

$$U_L = U_{C_L} + \frac{1}{2} L(\Delta U_L, \Delta U_R), \quad U_R = U_{C_R} - \frac{1}{2} L(\Delta U_L, \Delta U_R)$$

where

$$\Delta U_L = U_{C_L} - U_{P_L}, \quad \Delta U_R = U_{P_R} - U_{C_R}$$

and $L(\cdot, \cdot)$ is a limited average function. In the present work we have taken a min-mod type limiter [8] given by

$$L(a,b) = \frac{1}{2}(a+b)[1 - R(a,b)], \quad R(a,b) = \left| \frac{a-b}{\max(|a|+|b|, \epsilon)} \right|^3$$

The factor of "1/2" in the reconstruction formula is due to the fact that the centroid divides the line joining any vertex to the mid-point of the opposite side in the ratio 2:1.

The vertex values are obtained by using a weighted-inverse distance averaging

$$U_p = \frac{\sum_i \frac{w_{pi}}{r_{pi}} U_i}{\sum_i \frac{w_{pi}}{r_{pi}}}$$

where r_{pi} is the distance between vertex P and the centroid of a neighbouring triangle T_i which has P as a vertex, see fig. (17), the sum being over all such triangles. The factors w_{pi} ensure that the averaging formula has linear consistency, i.e., it is exact for linear functions and hence is second order accurate. For a boundary vertex, the triangle centroids are reflected about the tangent and the augmented set of vertices are used to compute the weights. This reflection step is used to enforce the slip velocity condition by reversing the sign of the velocity component perpendicular to the tangent. The procedure for determining these weights is given in appendix (A).

The finite volume equations are solved using a matrix-free LUSGS scheme with spectral radius approximation for the jacobians.

5.2. Adjoint solver

The development of the adjoint iterative solver follows the procedure outlined before. An additional consideration that arises here is due to the presence of the vertex averaging step and the enforcement of the zero normal velocity condition for inviscid flows, i.e., the velocity obtained

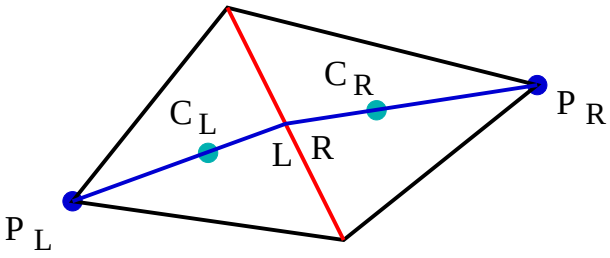


Figure 16: Reconstruction

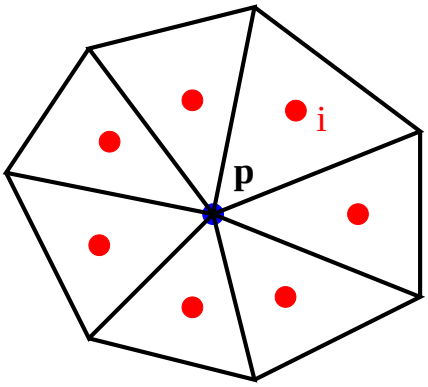


Figure 17: Vertex averaging

after averaging for a boundary vertex should not have any component normal to the boundary. In the flow solver it is more straight-forward to destroy the normal component after the averaging step. The flow solver has the following structure:

```
AveragingWeights
while the flow is not converged
   VertexAverage
   KillNormalVelocity
   ComputeFlux
   UpdateFlow
end
CostFunction
```

To generate the adjoint solver, each of the modules in the flow solver are differentiated in reverse mode with respect to the flow solution vector U. The adjoint iterative solver is constructed using the differentiated modules and has the following structure:

```
AveragingWeights
CostFunction_u
while the adjoint is not converged
   ComputeFlux_u
   KillNormalVelocity_u
   VertexAverage_u
   UpdateAdjoint
end
```

The suffix "_u" denotes that the modules are differentiated with respect to the flow solution. After the adjoint solution has converged the shape gradient $G = \frac{\partial J}{\partial \alpha} + v \left(\frac{\partial R}{\partial \alpha}\right)^\top$ is computed; the first term is easy to compute. For the second term the flow solver modules are differentiated in reverse mode with respect to the control variable. The structure of the code for computing the second term in the gradient has the following form:

```
ComputeFlux_x
KillNormalVelocity_x
VertexAverage_x
AveragingWeights_x
```

where the suffix "_x" denotes that the modules are differentiated with respect to some control variable. In the case of shape optimization the control variables are the coordinates of the grid points.

The flow solver code is written in multiple files which are compiled and linked using the `make` utility in Unix/Linux. The adjoint code is also compiled using `make` which automatically calls Tapenade to differentiate all the required subroutines. Any changes in the flow solver like the modification of some flux subroutine is immediately reflected in the adjoint solver by running `make`. The adjoint solver requires about 38% more memory compared to the flow solver. The cost per adjoint iteration is about twice the cost of a flow solver iteration.

5.3. Implicit scheme

The adjoint equations are solved by introducing a time derivative term and marching the adjoint solution in time using a Runge-Kutta scheme or an implicit scheme. The adjoint equation can be written as

$$A_i \frac{\mathrm{d}V_i}{\mathrm{d}t} + R_i^* = 0$$

The forward Euler implicit scheme with local time-stepping is given by

$$A_i \frac{V_i^{n+1} - V_i^n}{\Delta t_i} + R_i^*(V^{n+1}) = 0$$

The adjoint residual R^* has the form $Q+S$ where Q is linear in V and S contains the contribution from $\frac{\partial J}{\partial u}$ and is independent of the adjoint solution. The above equation can be written as

$$A_i \frac{V_i^{n+1} - V_i^n}{\Delta t_i} + Q_i(V^{n+1} - V^n) + R_i^*(V^n) = 0$$

Since we are only interested in the final solution obtained by driving the adjoint residual to zero, we can use a first order approximation for Q. This is given by

$$Q_i(V) = \sum_{j \in N(i)} \left\{ \left[\frac{\partial}{\partial X} F(U_i, U_j, \hat{n}_{ij}) \right]^\top V_i + \left[\frac{\partial}{\partial Y} F(U_j, U_i, \hat{n}_{ji}) \right]^\top V_j \right\} \Delta S_{ij}$$

The jacobian terms are still costly to evaluate; using the spectral radius approximation we obtain

$$\begin{aligned} \frac{\partial}{\partial X} F(U_i, U_j, \hat{n}_{ij}) &\approx \frac{1}{2}(A_i \cdot \hat{n}_{ij} + \lambda_{ij}) \\ \frac{\partial}{\partial Y} F(U_j, U_i, \hat{n}_{ji}) &\approx \frac{1}{2}(A_i \cdot \hat{n}_{ji} - \lambda_{ij}) \end{aligned}$$

where λ_{ij} is the maximum eigenvalue in the direction $\hat{n}_{ij}$ given by $\lambda_{ij} = |\vec{q}_i \cdot \hat{n}_{ij}| + c_i$ with $\vec{q}$ being the velocity vector and c the local speed of sound. With these approximations Q reduces to

$$Q_i(V) = \frac{1}{2} \left(\sum_{j \in N(i)} \lambda_{ij} \Delta S_{ij} \right) V_i - \frac{1}{2} \sum_{j \in N(i)} \left[A_i \cdot \hat{n}_{ij} + \lambda_{ij} \right]^\top V_j \Delta S_{ij}$$

where we have made use of the fact that $\sum_j \hat{n}_{ij} \Delta S_{ij} = 0$ and $\hat{n}_{ji} = -\hat{n}_{ij}$. The implicit equations are given by

$$\underbrace{\left(\frac{A_i}{\Delta t_i} + \frac{1}{2} \sum_{j \in N(i)} \lambda_{ij} \Delta S_{ij} \right)}_{D_i} \Delta V_i^n = -R_i^*(V^n) + \frac{1}{2} \sum_{j \in N(i)} \left[A_i \cdot \hat{n}_{ij} + \lambda_{ij} \right]^\top \Delta S_{ij} \Delta V_j^n$$

This is a set of coupled linear equations which can be solved using an iterative scheme like Gauss-Siedel, Gauss-Jacobi, LUSGS, etc. In the present work we have used the LUSGS scheme which is a two-step procedure and can be written as

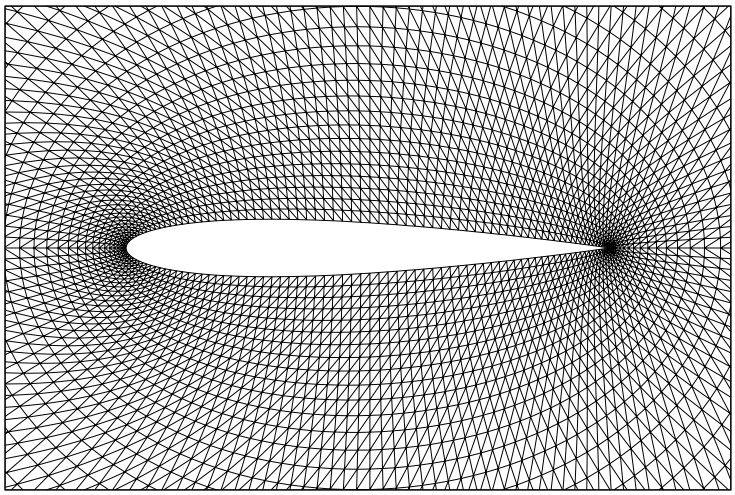


Figure 18: Triangular grid for NACA-0012 airfoil

1. Forward sweep: for $i = 1, 2, \ldots, N$

$$\Delta V_i^* = \frac{1}{D_i} \left\{ -R_i^*(V^n) + \frac{1}{2} \sum_{\substack{j \in N(i) \\ j<i}} \left[A_i \cdot \hat{n}_{ij} + \lambda_{ij} \right]^\top \Delta S_{ij} \Delta V_j^* \right\}$$

2. Reverse sweep: for $i = N, N-1, \ldots, 1$

$$\Delta V_i^n = \frac{1}{D_i} \left\{ D_i \Delta V_i^* + \frac{1}{2} \sum_{\substack{j \in N(i) \\ j>i}} \left[A_i \cdot \hat{n}_{ij} + \lambda_{ij} \right]^\top \Delta S_{ij} \Delta V_j^n \right\}$$

The local time step is calculated based on a CFL condition with the CFL number being of the order of 1000. In the present work flow over NACA-0012 airfoil is used for validating the adjoint solver. A grid consisting of 12800 triangles obtained by triangulating a structured grid is used in the computations. A zoomed view of the grid is shown in fig. (18). A sample of the adjoint solution corresponding to the density equation is shown in fig. (19); here the cost function is taken to be the lift coefficient. The convergence history of flow and adjoint iterations using implicit scheme is shown in fig. (20); we note that the flow and adjoint iterations have similar convergence rates.

5.4. Validation of gradients

The adjoint solution is again indirectly validated, by computing the derivative of lift coefficient with respect to angle of attack. The angle of attack is changed by rotating the grid. At subsonic flow and small angles of attack, the variation of lift with angle of attack is linear. The lift

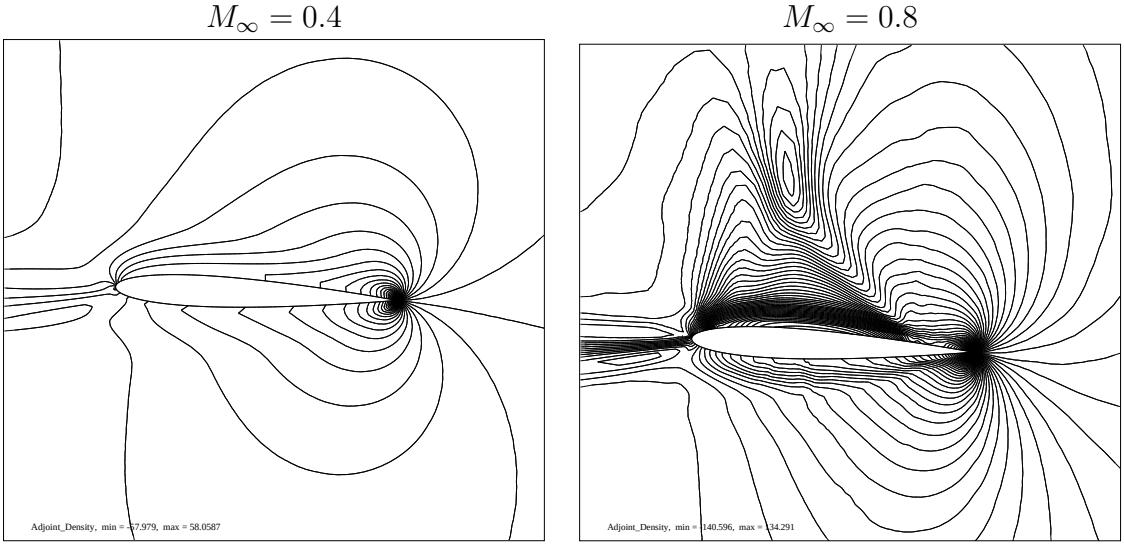


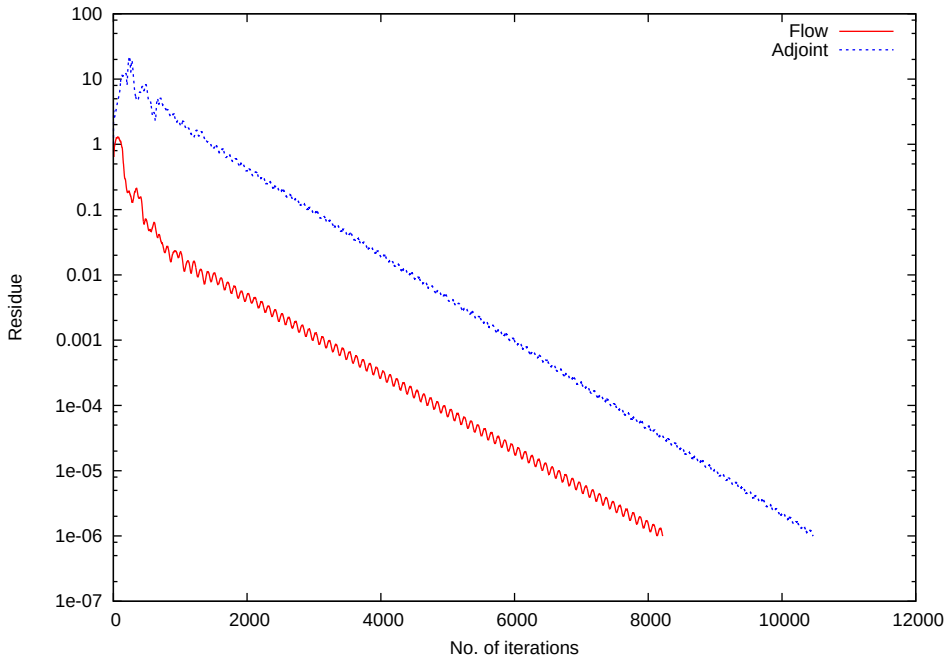
Figure 19: Adjoint variable corresponding to density equation for lift coefficient at AOA=3^o using kinetic flux

coefficient is computed at a free-stream Mach number of 0.4 and several angles of attack and these are plotted in fig. (21) as symbols. The dotted line is obtained by joining the symbols by straight line segments. At each angle of attack the adjoint solution and the derivative of lift coefficient are obtained and the values are listed in table (2) under the column C_{l_α}. In fig (21) a short line segment is drawn at each symbol with the slope given by the corresponding value of C_{l_α}. We note first of all that the slope given by AD is nearly constant and it agrees quite well with the linear variation of the lift coefficient[1]. The lift-curve slope is also compared to the theoretical value with Prandtl-Glauert correction given by $C_{l_\alpha}^{\text{th}} = 2\pi/\sqrt{1-M_\infty^2}$ and the ratio is listed in table (2). These results confirm the correctness of the adjoint solution. The same computations are also performed at a free stream Mach number of 0.8 and the computed derivatives are also indicated in table (2). Figure (22) shows the variation of the lift coefficient and in this case we notice that the behaviour is not linear[2]. However the computed slope agrees quite well with the lift-curve slope. The deviation from the theoretical value is much larger which is to be expected due to the presence of shocks.

The above computations are repeated with Roe flux [14] and the computed derivatives are listed in table (3). For subsonic flow, we note that the slope agrees quite well with the values obtained using kinetic flux, with the lift variation being nearly linear. For transonic flow, the lift-curve slope shows considerable difference from the values of kinetic flux. Also we notice an oscillatory variation in the slope which is also evident in fig. (24). Since limiter is used in both cases (Roe and kinetic flux) it appears that the flux function is responsible for this anamolous behaviour. The kinetic flux is infinitely differentiable while the Roe flux is not. This non-differentiability of Roe flux combined with the presence of shocks in the flow might be responsible for the oscillatory nature of the results.

[1]This can be checked by placing a ruler over the figure.
[2]Again, use a ruler to verify this.

$M_\infty = 0.4$



$M_\infty = 0.8$

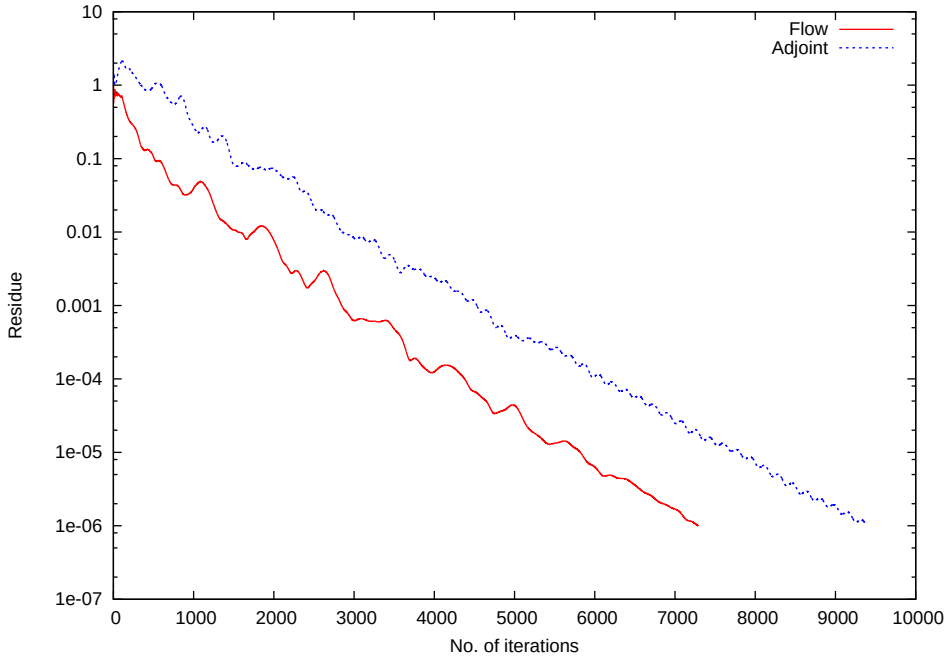


Figure 20: Convergence history of flow and adjoint solution using kinetic flux and LUSGS scheme

AOA	$M_\infty = 0.4$		$M_\infty = 0.8$	
	C_{l_α}	$C_{l_\alpha}/C_{l_\alpha}^{\text{th}}$	C_{l_α}	$C_{l_\alpha}/C_{l_\alpha}^{\text{th}}$
0.0	7.2095	1.0516	13.7506	1.3130
0.5	7.2069	1.0512	14.0556	1.3422
1.0	7.2000	1.0502	13.9544	1.3325
1.5	7.1901	1.0487	13.2116	1.2616
2.0	7.1789	1.0471	12.8314	1.2253
2.5	7.1675	1.0455	11.4077	1.0893
3.0	7.1562	1.0438	10.7135	1.0230

Table 2: Lift-curve slope with kinetic flux

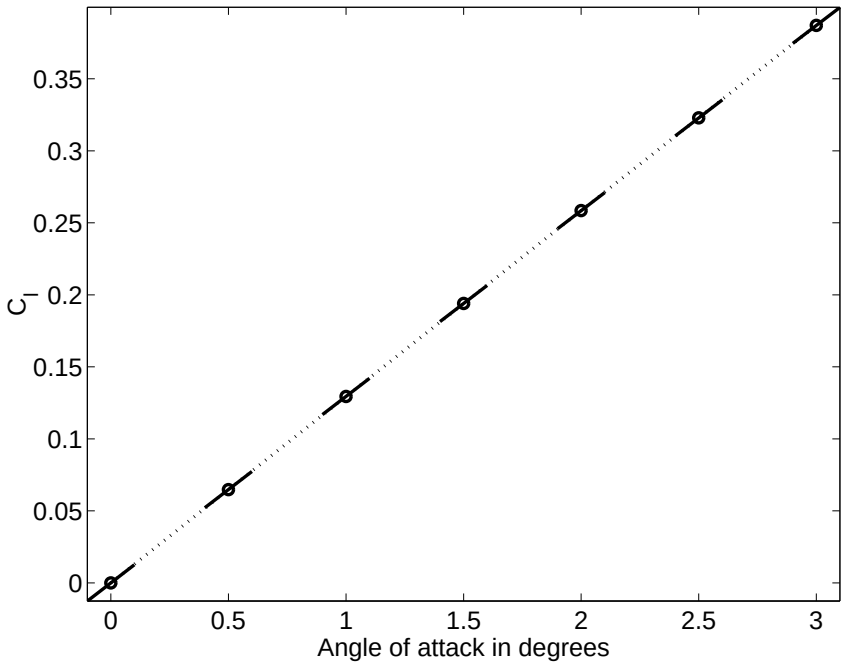


Figure 21: Lift curve for subsonic flow with kinetic flux

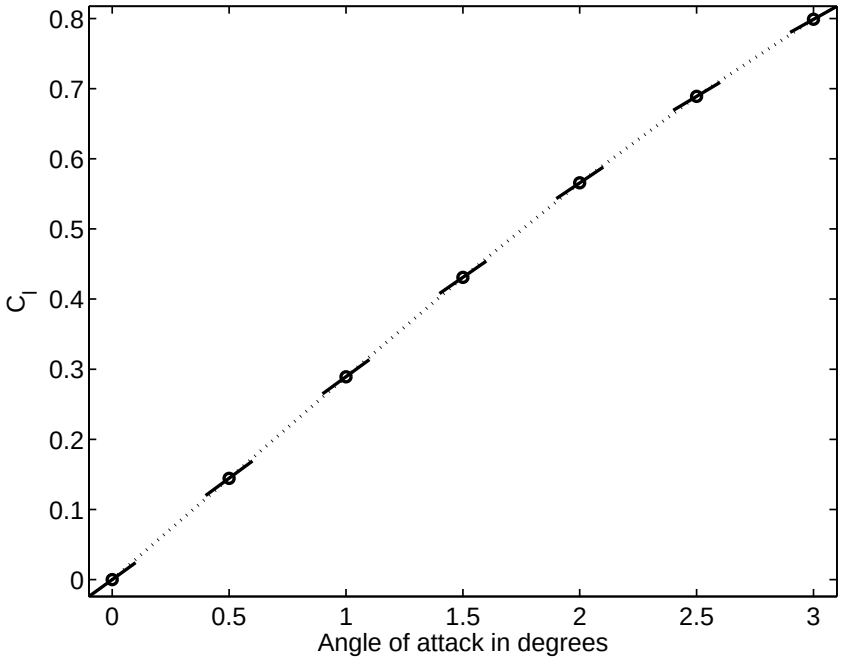


Figure 22: Lift curve for transonic flow with kinetic flux

AOA	$M_\infty = 0.4$		$M_\infty = 0.8$	
	C_{l_α}	$C_{l_\alpha}/C_{l_\alpha}^{\text{th}}$	C_{l_α}	$C_{l_\alpha}/C_{l_\alpha}^{\text{th}}$
0.0	7.3209	1.0678	8.8250	0.8427
0.5	7.3109	1.0664	9.6086	0.9175
1.0	7.2986	1.0646	10.7504	1.0265
1.5	7.2925	1.0637	10.1796	0.9720
2.0	7.2882	1.0631	9.4832	0.9055
2.5	7.2754	1.0612	8.2931	0.7919
3.0	7.2590	1.0588	16.1188	1.5392

Table 3: Lift-curve slope with Roe flux

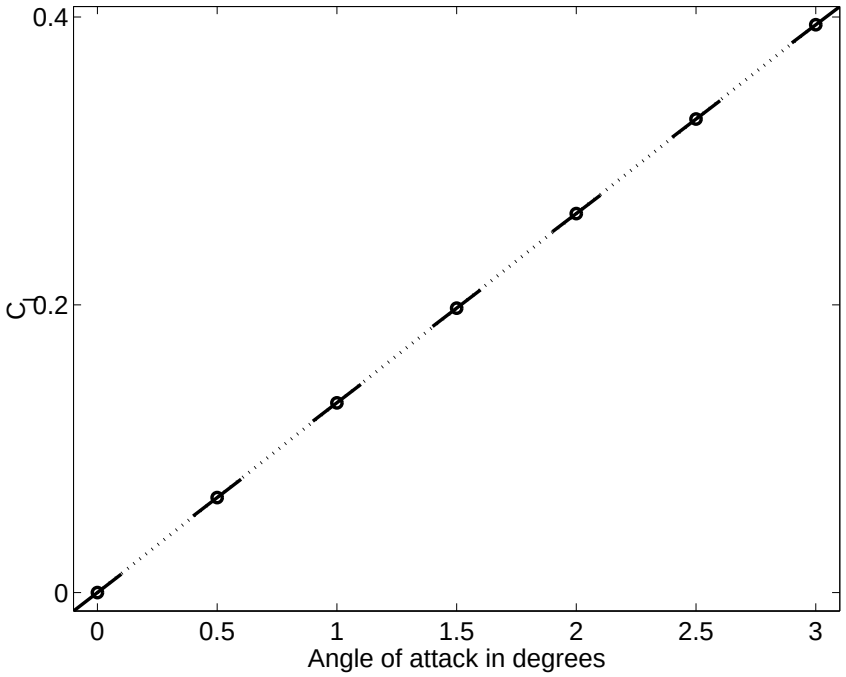


Figure 23: Lift curve for subsonic flow with Roe flux

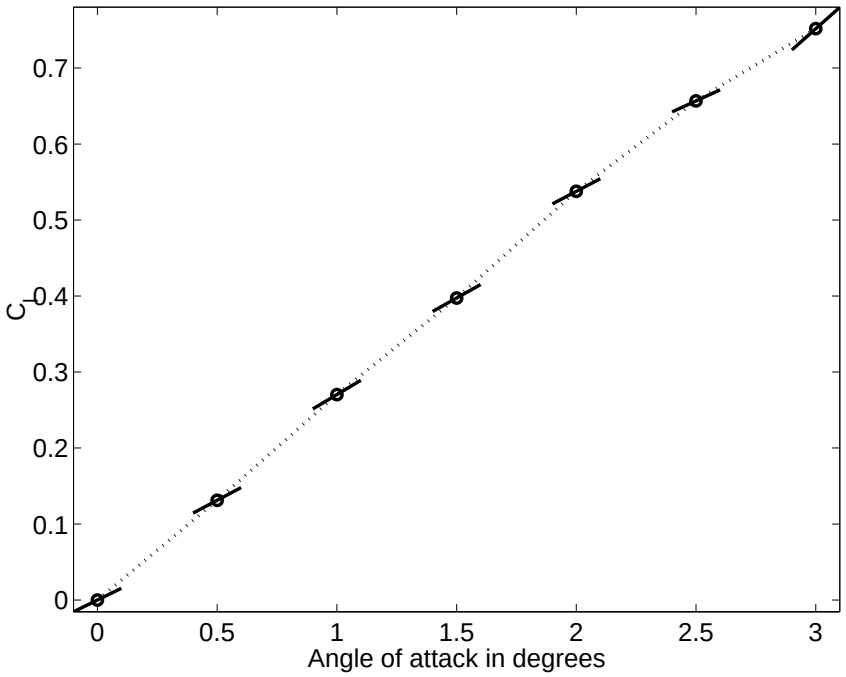


Figure 24: Lift curve for transonic flow with Roe flux

6. Summary

An automatic differentiation tool called Tapenade has been used to generate adjoint solvers for quasi 1-D and 2-D inviscid compressible flows modeled by the Euler equations. The adjoint solver is generated by applying AD in a piecemeal manner which leads to large savings in memory. The cost of adjoint iterations is also quite small being about twice of the flow solver. Both the flow and adjoint solvers are accelerated using LUSGS scheme. The gradients given by the adjoint solver are found to lack smoothness especially when the flow itself is discontinuous. Hence a smoothing procedure based on an elliptic equation is developed and applied to the computed derivatives before they are used in an optimization procedure. The computed gradients are validated by comparing with finite difference approximations. In 2-D a second order vertex-centroid scheme on triangular grids is used to develop the adjoint solver. This is validated by computing the lift-curve slope at subsonic condition and comparing with the expected linear behaviour.

A. Computation of weights for vertex averaging formula

The vertex-centroid scheme requires the solution values at the vertices also apart from the centroid values. The vertex values can be obtained by some interpolation procedure, the common ones being least squares, area-weighted averaging and inverse distance-weighted averaging. The accuracy of the least squares approach can in principle be increased arbitrarily; a second order finite volume scheme would require a least squares procedure with linear consistency, i.e., it should reproduce linear functions exactly. The cost of least squares interpolation is however high and/or requires more memory depending on whether one stores the coefficients of the interpolation formula or not. The averaging procedures based on area or distance averaging are cheaper but suffer from loss of accuracy on non-uniform grids; the area-weighted interpolation has linear consistency only if the vertex is at the centroid of the surrounding polygon. An averaging formula can be constructed with linear consistency by appropriately choosing the weights w,

$$U_p = \frac{\sum_i w_{pi} U_{pi}}{\sum_i w_{pi}}$$

where the weights satisfy

$$\sum_i w_{pi} \Delta x_{pi} = 0, \quad \sum_i w_{pi} \Delta y_{pi} = 0$$

This is usually an under-determined system and the weights are obtained by solving the following minimization problem:

$$\min_w \frac{1}{2} \sum_i (w_{pi} - 1)^2, \quad \text{subject to} \quad \sum_i w_{pi} \Delta x_{pi} = 0, \quad \sum_i w_{pi} \Delta y_{pi} = 0$$

The above constrained problem can be solved using Lagrange multipliers

$$\min_{w,\lambda_1,\lambda_2} \frac{1}{2} \sum_i (w_{pi} - 1)^2 + \lambda_1 \sum_i w_{pi} \Delta x_{pi} + \lambda_2 \sum_i w_{pi} \Delta y_{pi}$$

The first order conditions for the extremum lead to

$$w_{pi} = 1 - \lambda_1 \Delta x_{pi} - \lambda_2 \Delta y_{pi}$$

and

$$\begin{bmatrix} \sum \Delta x_{pi}^2 & \sum \Delta x_{pi}\Delta y_{pi} \\ \sum \Delta x_{pi}\Delta y_{pi} & \sum \Delta x_{pi}^2 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} \sum \Delta x_{pi} \\ \sum \Delta y_{pi} \end{bmatrix}$$

The matrix equation for (λ_1, λ_2) can be solved provided the vertex and the neighbouring centroids do not lie on a straight line. The weights can then be computed whenever required using (λ_1, λ_2) and there is no need to compute and store them in advance but it is enough to store (λ_1, λ_2) which is two real numbers per vertex. This procedure does not guarantee that the weights w are positive. If any of the weights is negative then the interpolating formula is not a convex combination; this can lead to instability since the interpolated value may not lie between the minimum and maximum of the neighbouring states.

In the present work we modify the averaging formula by incorporating inverse distance weighting

$$U_p = \frac{\sum_i \frac{1}{r_{pi}} w_{pi} U_{pi}}{\sum_i \frac{1}{r_{pi}} w_{pi}}$$

where r_{pi} is the distance between vertex P and the centroid of the a neighbouring triangle T_i. The weights are again determined by solving a minimization problem:

$$\min_w \frac{1}{2} \sum_i (w_{pi} - 1)^2, \quad \text{subject to} \quad \sum_i \frac{1}{r_{pi}} w_{pi} \Delta x_{pi} = 0, \quad \sum_i \frac{1}{r_{pi}} w_{pi} \Delta y_{pi} = 0$$

The solution of the above problem using Lagrange multipliers is given by

$$w_{pi} = 1 - \lambda_1 \cos(\theta_{pi}) - \lambda_2 \sin(\theta_{pi})$$

and

$$\begin{bmatrix} \sum \cos^2(\theta_{pi}) & \sum \cos(\theta_{pi}) \sin(\theta_{pi}) \\ \sum \cos(\theta_{pi}) \sin(\theta_{pi}) & \sum \sin^2(\theta_{pi}) \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} \sum \cos(\theta_{pi}) \\ \sum \sin(\theta_{pi}) \end{bmatrix}$$

where θ_{pi} is the angle made by the line joining vertex P and the centroid with the x-axis. Note that the equations for determining w, λ_1, λ_2 are independent of spatial scale. In practice we have found that the weights determined by this formula are *more* positive than any of the other formulae; also the weights are generally much closer to unity compared to the other formulae. A systematic study of the benefits of this averaging formula is yet to be conducted.

References

[1] `http://www-sop.inria.fr/tropics/tapenade.html`

[2] W. K. Anderson, J. C. Newman, D. L. Whitfield and E. J. Nielsen, "Sensitivity analysis for the Navier-Stokes equations on unstructured meshes using complex variables", AIAA-99-3294.

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

[4] N. T. Frink, "Recent progress toward a three-dimensional unstructured Navier-Stokes flow solver", AIAA-94-0061, 1994.

[5] M. B. Giles, D. Ghate and M. C. Duta, "Using automatic differentiation for adjoint CFD code development", Proc. of Post-SAROD Workshop, Tata McGraw Hill, 2005.

[6] L. Hascoet, "TAPENADE: a tool for Automatic Differentiation of programs", Proceedings of the ECCOMAS conference, Jyvaskyla, Finland, July 2004.

[7] L. Hascoet and V. Pascual, "TAPENADE 2.1 user's guide", INRIA Project Report No. 0300, September 2004.

[8] A. Jameson and John C. Vassberg, "A vertex-centroid (V-C) scheme for the gas-dynamics equations", In *Computational Fluid Dynamics (2000)*, ed. N. Satofuka, Springer, 2000.

[9] Manoj T. Nair., "Development of a two-dimensional aerodynamic optimization code based on sensitivity analysis", Project Document CF-0508, National Aerospace Laboratories, Bangalore, June 2005.

[10] J. C. Mandal and S. M. Deshpande, "Kinetic Flux Vector Splitting for Euler Equations", Computers and Fluids, vol 23, No. 2, pp. 447-478, 1994.

[11] Brian J. McCartin, "Seven Deadly Sins of Numerical Computation", American Math. Monthly, Vol. 105, No. 10, pp. 929-941, Dec. 1988.

[12] B. Mohammadi and O. Pironneau, *Applied shape optimization for fluids*, Clarendon Press, Oxford, 2001.

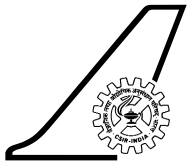
[13] E. J. Nielsen and W. K. Anderson, "Recent improvements in aerodynamic design optimization on unstructured meshes", AIAA Paper No. 2001- 0596, 2001.

[14] P. L. Roe, "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes", JCP, Volume 43, Number 2, October 1981, pages 357 372.

[15] Yasuhiro Wada and Meng-Sing Liou, "An Accurate and Robust Flux Splitting Scheme for Shock and Contact Discontinuities", SIAM Journal on Scientific Computing, Volume 18, Issue 3, pp. 633 - 657, 1997

[16] P. Wesseling, *Principles of Computational Fluid Dynamics*, Springer.

Documentation Sheet



National Aerospace Laboratories

Class Unrestricted

No. of Copies 10

Title	Adjoint code development and optimization using automatic differentiation
Author(s)	Praveen. C

Division	CTFD	**Project No.**	I-888-1/No. 19
Document No.	PD CF 0604	**Date of Issue**	April, 2006

Contents	37 Page(s)	28 Figure(s)	3 Table(s)	16 Reference(s)

External Participation None

Sponsor	None
Approval	Head, CTFD Division
Remarks	None
Keywords	Automatic Differentiation, TAPENADE, Adjoint solver, Optimization

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.