Abstract

Gradient-based optimization is useful for designing engineering systems and for improving existing designs. The efficient computation of gradients for systems governed by partial differential equations is non-trivial. Different approaches to compute gradients are discussed including adjoint equations and automatic differentiation.

An automatic differentiation tool called TAPENADE is used to compute sensitivities for optimization. The tool supports both direct and reverse modes of differentiation. In the reverse mode, a compute-store strategy is used which leads to large storage requirements. This can however be minimized by making small changes to the primal code. The tool is first applied to some test codes to illustrate the idea of automatic differentiation. Two optimization problems are solved. The first problem involves a quasi-one dimensional flow in a nozzle and the problem of pressure matching is solved using the direct mode. The second problem involves a control problem for the 1-D Burgers equations which is solved in the reverse mode. Both the problems illustrate that inverse problems can have non-unique solutions.

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

Shape optimization is assuming an important role in the design of critical components of an aircraft like the wing and fuselage. With the developments in numerical algorithms and computer hardware, it is now possible to solve optimization problems on the computer. Mathematically optimization problems can be cast as minimization problems. For example, one may wish to minimize the drag of a wing while keeping the lift coefficient constant. There are two approaches to solving a minimization problem; those which do not use a gradient like genetic algorithms, neural networks, etc., and those which make use of gradients. Gradient-free methods are easy to implement but are computationally expensive since they have slow convergence properties. However an important advantage of these methods is that they may be able to find the global minimum which may be difficult with gradient-based methods. This makes them ideal for preliminary design.

Gradient-based methods use some form of steepest descent algorithm to find the optimum state [5]. The major task in this approach is the accurate and efficient evaluation of gradients of the cost function with respect to the design variables.

The outline of the report is as follows. The formulation of an optimization problem is given. Two approaches, direct and adjoint, for evaluating gradients are described. Different methods for numerically computing the gradient, including automatic differentiation, are discussed. An automatic differentiation tool called TAPENADE is used. Many issues arising in the use of AD for solving CFD problems are highlighted with examples. Two model problems are solved using TAPENADE . The first one is the design of a quasi-1D nozzle for a given pressure distribution using direct method. The second problem involves a control problem governed by the 1-D Burgers equations. Both examples illustrate the non-uniqueness of solutions to inverse problems.

2. Minimization problem

The objective is to minimize a *cost function* $I(\alpha, u(\alpha))$ where α is the *control* or *design variable* and u is a *state variable* which is governed by a partial differential equation like the Euler or Navier-Stokes equations. The minimization problem can be stated as

$$\min_{\alpha} I(\alpha, u(\alpha)) \tag{1}$$

subject to state constraint (which includes the boundary conditions also)

$$R(\alpha, u(\alpha)) = 0 \tag{2}$$

and some geometric constraints

$$g(\alpha) \leq 0 \tag{3}$$

This is an example of a constrained minimization problem. The functions above I, R, g could be continuous functions, for example partial differential equations or they could be discrete versions obtained by applying a suitable discretization scheme for the partial differential equations.

2.1. Direct method

Differentiating the cost function, we get

$$\frac{\mathrm{d}I}{\mathrm{d}\alpha} = \frac{\partial I}{\partial \alpha} + \frac{\partial I}{\partial u}\frac{\partial u}{\partial \alpha} \tag{4}$$

where $\partial u/\partial \alpha$ is called the *flow sensitivity* which must be obtained from the state equation. Differentiating the state equation

$$\frac{\partial R}{\partial \alpha} + \frac{\partial R}{\partial u}\frac{\partial u}{\partial \alpha} = 0 \tag{5}$$

or

$$\frac{\partial R}{\partial u}\frac{\partial u}{\partial \alpha} = -\frac{\partial R}{\partial \alpha} \tag{6}$$

which is a linear equation and is in fact a linearization of the state equation around some given state. The flow sensitivity can be obtained in several ways.

1. By solving the sensitivity equations (6)
2. By using finite differences
3. By using complex Taylor series expansion
4. By automatic differentiation

These approaches will be discussed in the next section. The direct approach is costly if the number of control variables is large since it requires one linearized state (Euler or Navier-Stokes) solution for each control variable. The adjoint approach overcomes this problem.

2.2. Adjoint method

The basic idea in adjoint method is to eliminate direct computation of flow sensitivities. The state constraint can be included in the cost function using Lagrange multipliers

$$J(\alpha, \lambda) = I(\alpha, u(\alpha)) + \lambda^{\top} R(\alpha, u(\alpha)) \tag{7}$$

where the second term on the right hand side is an inner product. The minimization problem is now stated as

$$\min_{\alpha,\lambda} J(\alpha, \lambda) \tag{8}$$

which is an unconstrained problem except for geometric constraints. Differentiating the new cost function

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial I}{\partial \alpha} + \frac{\partial I}{\partial u}\frac{\partial u}{\partial \alpha} + \lambda^{\top}\left(\frac{\partial R}{\partial \alpha} + \frac{\partial R}{\partial u}\frac{\partial u}{\partial \alpha}\right) \tag{9}$$

or rearranging the terms

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial I}{\partial \alpha} + \lambda^{\top}\frac{\partial R}{\partial \alpha} + \left(\frac{\partial I}{\partial u} + \lambda^{\top}\frac{\partial R}{\partial u}\right)\frac{\partial u}{\partial \alpha} \tag{10}$$

Now, if we choose the Lagrange multiplier λ which is also called the *adjoint variable* such that

$$\frac{\partial I}{\partial u} + \lambda^\top \frac{\partial R}{\partial u} = 0$$

or

$$\left(\frac{\partial R}{\partial u}\right)^\top \lambda = -\left(\frac{\partial I}{\partial u}\right)^\top \tag{11}$$

then the gradient of the cost function becomes

$$\frac{\mathrm{d}J}{\mathrm{d}\alpha} = \frac{\partial I}{\partial \alpha} + \lambda^\top \frac{\partial R}{\partial \alpha} \tag{12}$$

which does not involve the flow sensitivity. Equation (11) is called the *adjoint equation*. The evaluation of $\partial I/\partial\alpha$ and $\partial R/\partial\alpha$ is cheap compared to the evaluation of the flow sensitivities and the adjoint solution. Note that the right hand side of the adjoint equation depends on the cost function (actually on its derivative). Thus the adjoint equation depends on the cost function, and the dependence either appears on the right hand side of the adjoint equation or in the boundary conditions for the adjoint equation.

The cost of evaluating the gradients with the adjoint method is nearly independent of the number of control variables since it requires only one adjoint solution (whose cost is nearly equivalent to that of one state solution). Hence the adjoint approach is preferred over the direct approach when the number of control/design variables are large.

The adjoint equations can be derived either at the *continuous* partial differential equation level (differentiate-discretize approach) which leads to a set of linear partial differential equations or at the *discrete* level (discretize-differentiate approach), which leads to a set of linear algebraic equations [9]. The continuous approach has been pioneered by Jameson and co-workers [12, 13, 14], while the discrete approach has been used in [2, 3, 4, 7, 8, 18, 19]. The continuous approach has some drawbacks.

- The adjoint equations and boundary conditions have to be derived whenever the cost function is changed. Moreover not all cost functions lead to a well-posed adjoint problem.
- The solution of the adjoint equations is obtained by solving them numerically. But this solution does not give the true sensitivity of the state solution since it does not account for numerical dissipation inherent in the solution of the state equation.

The discrete approach does not have these drawbacks; there is no need to explicitly derive boundary conditions and the adjoint solution gives the true sensitivity of the discrete state solution.

3. Gradient estimation techniques

Accurate and efficient estimation of the gradients is the most crucial part of an optimization process. This is particularly true when a large number of design variables are present. Different methods for estimating gradients with respect to the design variables are discussed.

3.1. Sensitivity equations

In this approach, the sensitivity equations are derived starting from the state equation, which is usually a partial differential equation. For simplicity, consider a linear PDE

$$Au = f \text{ in } \Omega \tag{13}$$

with boundary condition

$$u = g \text{ on } \partial\Omega \tag{14}$$

Assuming that only f and g depend on the control variable α, we obtain after differentiation

$$\begin{aligned} Au_\alpha &= f_\alpha \text{ in } \Omega \\ u_\alpha &= g_\alpha \text{ on } \partial\Omega \end{aligned}$$

where u_α is the flow sensitivity. The sensitivity equations are discretized using an appropriate numerical scheme and solved on a computer. Even though the sentivity equations are linear they can be rather stiff leading to slow convergence. If there are N_c control variables then we have to solve N_c sensitivity equations which can be computationally expensive if N_c is large.

3.2. Finite differences

This is the simplest method of evaluating gradients and relies on finite differences. If there are N_c control variables then,

`For` $j = 1, N_c$

1. `Choose a step size` $\Delta\alpha_j$
2. `Evaluate the state with control` α
3. `Evaluate the state with control`[1] $\alpha + e_j\Delta\alpha_j$
4. `Use finite difference`

$$\frac{\partial u}{\partial \alpha_j} \approx \frac{u(\alpha + e_j\Delta\alpha_j) - u(\alpha)}{\Delta\alpha_j} \tag{15}$$

This method requires $N_c + 1$ state evaluations and is hence computationally expensive if N_c is large. It is useful when the state solver is not available in source code form but only as an executable. The accuracy of the gradient is highly dependent on the step size; if a very small step size is taken then the method suffers from cancellation errors since we are subtracting nearly equal quantities. If the step size is large then truncation errors will dominate and the accuracy will again suffer. There is an optimum value of step size which gives least error [17] but this is difficult to determine in practical situations.

[1] e_j is a vector which has one in the j'th position and zero elsewhere.

3.3. Complex variables

The complex variable method eliminates the cancellation error inherent in the finite difference method. Here, we make a small complex perturbation to the current design variables and use Taylor series to obtain

$$u(\alpha + \mathrm{i}e_j\Delta\alpha_j) = u(\alpha) + \mathrm{i}\Delta\alpha_j \frac{\partial u}{\partial \alpha_j} + O(\Delta\alpha^2) + \mathrm{i}O(\Delta\alpha^3)$$

Taking the imaginary part

$$\mathrm{imag}[u(\alpha + \mathrm{i}e_j\Delta\alpha_j)] = \Delta\alpha_j \frac{\partial u}{\partial \alpha_j} + O(\Delta\alpha^3)$$

Hence we have

$$\frac{\partial u}{\partial \alpha_j} = \frac{\mathrm{imag}[u(\alpha + \mathrm{i}e_j\Delta\alpha_j)]}{\Delta\alpha_j} + O(\Delta\alpha^2)$$

Note that there is no cancellation of nearly equal terms like in the finite difference method and also the approximation is second order accurate. The cost of this method is however slightly more than that of the finite difference method due to the need for complex arithmetic. A very small value of the step size, of the order of 10^{-20} can be used without any problem of round-off error [9]. When the number of design variables is small this method is ideal and is to be preferred over the finite difference method. Its implementation is also very simple since we have to just declare all variables as complex variables. Anderson et al. [4] have used complex variables to compute the sensitivities for turbulent flows and found that they yield results as accurate as automatic differentiation and are superior to finite difference results. However the memory requirements for complex variable method is essentially double of the original code and the computation time can increase by as much as a factor of three [4]. Nair [16] has used this approach to compute the jacobian terms arising in the adjoint equation (11) and used it to solve pressure-matching problems for 2-D inviscid flows.

3.4. Automatic differentiation

In the discretize-differentiate approach, we can differentiate the computer code for solving the state equation *manually* and generate a computer code for solving the sensitivity equations. However this process is laborious, time-consuming and prone to errors. Since a numerical scheme is made up of a small set of standard functions and the usual rules of arithmetic it should be possible to automate the process of differentiating an existing computer code. This is what automatic differentiation (AD) refers to. There are many tools available today which implement automatic differentiation like ADIFOR, ADOLC, TAMC, TAF and TAPENADE, see [1]. Most of these tools take a computer source code as input and generate another source code which evaluates the gradients. AD is not like symbolic differentiation which is performed by softwares like Maxima and Mathematica; AD mainly uses the chain rule of differentiation and the output of an AD tool is a computer code which is much more useful than the output of a symbolic differentiation tool. Moreover AD tools also provide an adjoint approach which is not possible with symbolic differentiation tools.

There are two approaches to automatic differentiation corresponding to the two approaches discussed before: direct method of sensitivity computation and reverse or adjoint method. The

direct method is a straight-forward application of the chain rule of differentiation and is discussed first.

3.4.1. Direct method

Let us look at an example which has two independent variables u_1, u_2 and a single cost variable J.

$$\begin{aligned} y_1 &= u_1^2 + 5u_2 \\ y_2 &= u_1 y_1 + u_2^3 \\ J &= u_1 + u_2 + y_1 y_2 \end{aligned}$$

Note that there are two intermediate variables y_1, y_2 and it is required to find the gradient of J with respect to (u_1, u_2).

```
C This subroutine takes u1, u2 as input and returns J
      subroutine costfunc(u1, u2, J)
      implicit none
      real y1, y2, u1, u2, J

      y1 = u1**2 + 5.0*u2
      y2 = u1*y1 + u2**3
      J  = u1 + u2 + y1*y2

      return
      end
```

The differented code using TAPENADE is given below[2].

```
      SUBROUTINE COSTFUNC_D(u1, u1d, u2, u2d, J, Jd)
      IMPLICIT NONE
      REAL J, Jd, u1, u1d, u2, u2d
      REAL y1, y1d, y2, y2d

      y1d = 2*u1*u1d + 5.0*u2d
      y1  = u1**2 + 5.0*u2
      y2d = u1d*y1 + u1*y1d + 3*u2**2*u2d
      y2  = u1*y1 + u2**3
      Jd  = u1d + u2d + y1d*y2 + y1*y2d
      J   = u1 + u2 + y1*y2

      RETURN
      END
```

[2]In the direct mode of TAPENADE , the differential of a variable `var` is denoted by `vard` ie., the letter `d` is added to the end of the variable name.

Note that the lines from the original program are present in the differentiated program. Above each line of the original code, a differentiated line has been added. If we call `COSTFUNC_D` with `(u1d=1, u2d=0)` then `Jd` will give $\partial J/\partial u_1$. Similarly, if we call `COSTFUNC_D` with `(u1d=0, u2d=1)` then `Jd` will give $\partial J/\partial u_2$. We can also obtain the derivative in any direction θ by setting `u1d=cos(theta), u2d=sin(theta)`. So the direct method gives the gradient in some specified direction. The disadvantage of this method is same as that of finite difference method; the program has to be run N_c times.

3.4.2. Reverse method

The reverse method of automatic differentiation is the analogue of the adjoint method. Let us illustrate this with an example. Assume that there are two input variables (which are also the design variables) u_1, u_2, two intermediate variables y_1, y_2 and a single cost function J which depends on the input and the intermediate variables.

$$\begin{aligned} y_1 &= l_1(u_1, u_2) \\ y_2 &= l_2(u_1, u_2, y_1) \\ J &= l_3(u_1, u_2, y_1, y_2) \end{aligned}$$

Note that this corresponds to the example considered in the last section.

Recipe for reverse method:

1. With each intermediate variable y_i we associate a dual or adjoint variable p_i and form the Lagrangian
$$L(u, y, p) := p_1[y_1 - l_1(u)] + p_2[y_2 - l_2(u, y_1)] + J(u) - l_3(u, y) \tag{16}$$
2. Make the Lagrangian stationary with respect to y_2, y_1 (in that order) to obtain
$$0 = p_2 - \frac{\partial l_3}{\partial y_2}(u, y_1, y_2) \tag{17}$$
$$0 = p_1 - p_2 \frac{\partial l_2}{\partial y_1}(u, y_1) - \frac{\partial l_3}{\partial y_1}(u, y_1, y_2) \tag{18}$$
3. Solve the above equations for p_2 and p_1 respectively.
4. Make the Lagrangian stationary with respect to u_i to obtain the gradient of the cost function J
$$\frac{\partial J}{\partial u_i} = p_1 \frac{\partial l_1}{\partial u_i} + p_2 \frac{\partial l_2}{\partial u_i} + \frac{\partial l_3}{\partial u_i} \tag{19}$$

Note that if we make the Lagrangian stationary with respect to the adjoint variable then we get the equations for the intermediate variables. Let us apply the above procedure to the test problem considered in the previous section. We have

$$\begin{aligned} p_2 &= \frac{\partial l_3}{\partial y_2}(u, y_1, y_2) \\ &= y_1 \\ p_1 &= p_2 \frac{\partial l_2}{\partial y_1}(u, y_1) + \frac{\partial l_3}{\partial y_1}(u, y_1, y_2) \\ &= y_1 u_1 + y_2 \end{aligned}$$

Note that the adjoint variables are solved in reverse and hence this is known as reverse method. Using TAPENADE , the differentiated code is given below.

```
      SUBROUTINE COSTFUNC_B(u1, u1b, u2, u2b, J, Jb)
      IMPLICIT NONE
      REAL J, Jb, u1, u1b, u2, u2b
      REAL y1, y1b, y2, y2b

      y1 = u1**2 + 5.0*u2
      y2 = u1*y1 + u2**3

      y2b = y1*Jb
      y1b = u1*y2b + y2*Jb
      u1b = 2*u1*y1b + y1*y2b + Jb
      u2b = 5.0*y1b + 3*u2**2*y2b + Jb
      Jb = 0.0
      END
```

Note that the intermediate variables are again computed in the differentiated code. The variables `y1b, y2b` are the adjoint variables[3] corresponding to `y1, y2` respectively and we see that the adjoint variables are computed in reverse order. The gradients with respect to u_1, u_2 are given by `u1b, u2b` respectively. Since `J` is the dependent variable we have to call the above subroutine with `Jb=1`, i.e., the adjoint variable corresponding to the variable whose derivative is required must be set to unity (In the recipe given above we have already made this assumption). Comparing the above code with the exactly derived adjoints p_1, p_2 we see that the procedure outined above is followed in the differentiated code.

When the output or dependent variable is a vector $R(u)$, then the reverse mode returns $(\partial R/\partial u)^\top \lambda$ for a given vector λ. For example, this can be used to solve the adjoint equation (11) by writing a separate code which implements a relaxation scheme for the adjoint variable [6]. Note that this method also saves memory requirements since there is no need to store the matrix $\partial R/\partial u$ which can be very large in CFD problems.

4. TAPENADE: An automatic differentiation tool

TAPENADE is an automatic differentiation tool being developed at INRIA. It is a source tranformation tool which provides both the direct and reverse modes of differentiation and supports the FORTRAN language. The software is written in JAVA and executables are available for Linux, SUN and Windows platforms. A web-based tool is also available which can be run from an internet browser by visiting the following url,

`http://tapenade.inria.fr:8080/tapenade/`

Examples of the use of TAPENADE have been presented in the previous sections. Here we discuss some issues involved in the efficient use of TAPENADE particularly for CFD problems. A brief description about the use of TAPENADE is given in appendix (A).

[3] In TAPENADE , the adjoint variable corresponding to any variable `var` is named `varb`

4.1. Inter-procedural differentiation

Loops are very commonly used in numerical programs. If there is a lot of computation performed inside a loop then the loop contents should be put in a separate function or subroutine. Otherwise the memory requirements can be high since the intermediate values will have to be stored into the computers stack. As an example consider the function given in Fig. (1) where the independent variable is `x` and the dependent variable is `cost`. The differentiation of this program in reverse mode is shown in Fig. (2) and we notice the call to the function `PUSHREAL4` which saves the intermediate variables `y1, y2` into stack. These are then retrieved by the call to the function `POPREAL4` in subsequent portion of the program. In figure (3) the program has been modified by defining a subroutine `flux` and putting all the computations inside this subroutine. The differentiation of this program in reverse mode is given in figure (4) and we notice that there are no calls to `PUSH/POP` functions. Thus using inter-procedural differentiation can considerably reduce the memory requirements of the differentiated programs. This is very desirable in a CFD code which already has large memory overheads.

```
subroutine costfunc1(x, cost)
implicit none
real x(10), f(10), cost, y1, y2, y3
integer i
do i=1,10
  f(i) = 0.0
enddo
do i=1,9
  y1 = 0.5*( x(i) + x(i+1) )
  y2 = 0.25*( x(i)**2 + x(i+1)**2 )/y1
  y3 = 0.5*y1*( x(i+1) - x(i) ) + sin(y2)
  f(i) = f(i) + y2 - y3
  f(i+1) = f(i+1) - y2 + y3
enddo
cost = 0.0
do i=2,9
  cost = cost + f(i)**2
enddo
return
end
```

Figure 1: Example code to illustatrate inter-procedural differentiation

4.2. Over-writing intermediate variables

In a computer program, it is quite common to use intermediate variables to simplify some complex expression by breaking it into smaller expressions. In such cases the intermediate variables may be re-used to store different values at different stages of the code. This will cause

```
SUBROUTINE COSTFUNC1_B(x, xb, cost, costb)
IMPLICIT NONE
REAL cost, costb, x(10), xb(10)
INTEGER i, ii1
REAL f(10), fb(10), tempb, tempb0, y1, y1b, y2, y2b, y3, y3b
INTRINSIC SIN
DO i=1,10
  f(i) = 0.0
ENDDO
DO i=1,9
  CALL PUSHREAL4(y1)
  y1 = 0.5*(x(i)+x(i+1))
  CALL PUSHREAL4(y2)
  y2 = 0.25*(x(i)**2+x(i+1)**2)/y1
  y3 = 0.5*y1*(x(i+1)-x(i)) + SIN(y2)
  f(i) = f(i) + y2 - y3
  f(i+1) = f(i+1) - y2 + y3
ENDDO
DO ii1=1,10
  fb(ii1) = 0.0
ENDDO
DO i=9,2,-1
  fb(i) = fb(i) + 2*f(i)*costb
ENDDO
DO ii1=1,10
  xb(ii1) = 0.0
ENDDO
DO i=9,1,-1
  y3b = fb(i+1) - fb(i)
  y2b = COS(y2)*y3b + fb(i) - fb(i+1)
  tempb = 0.5*y1*y3b
  tempb0 = 0.25*y2b/y1
  y1b = 0.5*(x(i+1)+(-x(i)))*y3b - (x(i)**2+x(i+1)**2)*tempb0/y1
  xb(i+1) = xb(i+1) + tempb
  xb(i) = xb(i) + 2*x(i)*tempb0 - tempb
  CALL POPREAL4(y2)
  xb(i+1) = xb(i+1) + 2*x(i+1)*tempb0
  CALL POPREAL4(y1)
  xb(i) = xb(i) + 0.5*y1b
  xb(i+1) = xb(i+1) + 0.5*y1b
ENDDO
i = 0
costb = 0.0
END
```

Figure 2: Differentiation of code in Fig. (1) in reverse mode

```
subroutine costfunc2(x, cost)
implicit none
real x(10), f(10), cost, fl
integer i
do i=1,10
  f(i) = 0.0
enddo
do i=1,9
  call flux(x(i), x(i+1), fl)
  f(i) = f(i) + fl
  f(i+1) = f(i+1) - fl
enddo
cost = 0.0
do i=2,9
  cost = cost + f(i)**2
enddo
return
end

subroutine flux(x1, x2, fl)
implicit none
real x1, x2, fl, y1, y2, y3
y1 = 0.5*( x1 + x2 )
y2 = 0.25*( x1**2 + x2**2 )/y1
y3 = 0.5*y1*( x2 - x1 ) + sin(y2)
fl = y2 - y3
return
end
```

Figure 3: Modification of code in Fig. (1) with addition of function call within the loop

```
SUBROUTINE COSTFUNC2_B(x, xb, cost, costb)
IMPLICIT NONE
REAL cost, costb, x(10), xb(10)
REAL f(10), fb(10), fl, flb
INTEGER i, ii1
DO i=1,10
  f(i) = 0.0
ENDDO
DO i=1,9
  CALL FLUX(x(i), x(i+1), fl)
  f(i) = f(i) + fl
  f(i+1) = f(i+1) - fl
ENDDO
DO ii1=1,10
  fb(ii1) = 0.0
ENDDO
DO i=9,2,-1
  fb(i) = fb(i) + 2*f(i)*costb
ENDDO
DO ii1=1,10
  xb(ii1) = 0.0
ENDDO
DO i=9,1,-1
  flb = fb(i) - fb(i+1)
  CALL FLUX_B(x(i), xb(i), x(i+1), xb(i+1), fl, flb)
ENDDO
i = 0
costb = 0.0
END
```

Figure 4: Differentiation of code in Fig. (3) in reverse mode

TAPENADE to store the intermediate variables in a stack and use them back during the reverse mode.

Consider an example function shown in figure (5), in which an intermdiate variable name `tmp` is overwritten once.

```
subroutine costfunc(u1, u2, cost)
implicit none
real u1, u2, cost, tmp, y1, y2

tmp = u1 + u2
y1  = tmp*u1*u2
tmp = u1 - u2
y2  = tmp*u1*u2
cost= y1 + y2 + y1*u1 + y2*u2

return
end
```

Figure 5: Example function in which temporary variable `tmp` is over-written

The differentiated code in reverse mode is given in figure (6). Note that after the first initialization of `tmp` its value is stored into stack by a call to the `PUSHREAL4` function and it is later retrieved by a call to `POPREAL4` function.

In figure (7) the function has been modified by using two intermediate variables `tmp1` and `tmp2` so that there is no over-writing. The differentiated code is shown in figure (8) and we see that there is no PUSH/POP statement in this case. It is preferable to avoid calls to `PUSH/POP` since it slows down the code.

4.3. Function arguments

While calling a function, long expressions should be avoided in the function arguments. For example

```
c = EXP(a**2 + a*b + b**2)
```

should be replaced with

```
tmp = a**2 + a*b + b**2
c   = EXP(tmp)
```

The second choice is preferable since the argument might be required in the revese sweep of the adjoint method and there is no need to recalculate it again.

```
SUBROUTINE COSTFUNC_B(u1, u1b, u2, u2b, cost, costb)
IMPLICIT NONE
REAL cost, costb, u1, u1b, u2, u2b
REAL tmp, tmpb, y1, y1b, y2, y2b

tmp = u1 + u2
y1 = tmp*u1*u2
CALL PUSHREAL4(tmp)
tmp = u1 - u2
y2 = tmp*u1*u2

y1b = (u1+1.0)*costb
y2b = (u2+1.0)*costb
tmpb = u2*u1*y2b
u1b = tmpb + u2*tmp*y2b + y1*costb
u2b = tmp*u1*y2b - tmpb + y2*costb
CALL POPREAL4(tmp)
tmpb = u2*u1*y1b
u1b = u1b + tmpb + u2*tmp*y1b
u2b = u2b + tmpb + tmp*u1*y1b
costb = 0.0
END
```

Figure 6: Differentiation of function in figure (5) in reverse mode

```
subroutine costfunc(u1, u2, cost)
implicit none
real u1, u2, cost, tmp1, tmp2, y1, y2

tmp1 = u1 + u2
y1  = tmp1*u1*u2
tmp2 = u1 - u2
y2  = tmp2*u1*u2
cost = y1 + y2 + y1*u1 + y2*u2

return
end
```

Figure 7: Modification of function in figure (5); two temporary variables `tmp1` and `tmp2` are used.

```
SUBROUTINE COSTFUNC_B(u1, u1b, u2, u2b, cost, costb)
IMPLICIT NONE
REAL cost, costb, u1, u1b, u2, u2b
REAL tmp1, tmp1b, tmp2, tmp2b, y1, y1b, y2, y2b

tmp1 = u1 + u2
y1 = tmp1*u1*u2
tmp2 = u1 - u2
y2 = tmp2*u1*u2

y1b = (u1+1.0)*costb
y2b = (u2+1.0)*costb
tmp2b = u2*u1*y2b
tmp1b = u2*u1*y1b
u1b = tmp1b + u2*tmp1*y1b + tmp2b + u2*tmp2*y2b + y1*costb
u2b = tmp1b + tmp1*u1*y1b - tmp2b + tmp2*u1*y2b + y2*costb
costb = 0.0
END
```

Figure 8: Differentiation of function in figure (7) in reverse mode; Note that there is no PUSH/POP statement

4.4. Non-differentiable functions

Many functions like `min, max, abs` etc. are not differentiable everywhere. They usually appear in limiters and in some flux function formulae. First of all, TAPENADE replaces such functions by a simple `if` statement. For example

```
a = max(b, c)
```

is replaced by

```
if (b .lt. c) then
      a = c
else
      a = b
endif
```

The above code can be differentiated using one-sided derivatives and the result in direct mode is

```
if (b .lt. c) then
      ad= cd
      a = c
else
      ad= bd
      a = b
endif
```

It is preferable to define local implementations of such functions since the differentiated code is much more readable in that case.

The choice of the cost function must also be made with some care though non-differentiability in a cost function is not a very serious problem [15]. For example, while designing to match the pressure distribution it is better to take the cost function as

$$I = \int (p - p_d)^2 \mathrm{d}s$$

rather than

$$I = \left[\int (p - p_d)^2 \mathrm{d}s \right]^{1/2}$$

since the latter can lead to numerical problems as the design point is approached (since the gradient is of the form 0/0) while both the cost functions should theoretically lead to the same minimum solution.

4.5. Iterative loops

A typical CFD solver can be described by the following pseudo-code:

```
Read the grid file
Initalize the solution to free-stream values
do
      Calculate time step
      Calculate flux divergence
      Update the solution
      Find the residue
while(residue > MINRESIDUE)
Calculate cost function
```

Note that the solution is obtained from an iterative scheme[4] and details of the iteration and the intermediate solution are not required for calculating the cost function or its derivative. However TAPENADE will differentiate the iterative loop also[5] and will save all the intermediate solutions into the stack. This is not only unneccessary but also prohibitively expensive in terms of storage [15]. At present, the only way to eliminate this storage problem in TAPENADE is to manually remove all the unnecessary save statements from the differentiated code. This is not an ideal solution since it spoils the automatic nature of the process and is also prone to errors.

As an example consider the following cost function

$$J = ax$$

where x is the solution of the equation $f(x) = x^2 - a = 0$. This equation is solved iteratively using Newton-Raphson method. The code for calculating the cost function is given in figure (9). The subroutine `solve` implements the iterative loop and its differentiation in reverse mode is given in figure (10) where the forward and reverse loops are indicated. Note that in the forward

[4]For example, iterating to a steady state using local time-stepping

[5]TAMC and TAF can handle iterative loops.

loop the intermediate solution is stored into stack by the statement `CALL PUSHREAL4(x)` and this stored value is retrieved in the backward sweep by the statement `CALL POPREAL4(x)`. The differentiated code is modified in the following way and the result is shown in figure (11).

1. Remove the forward loop completely.
2. From the reverse loop, remove the POP statements.
3. Replace the `DO` loop with a conditional loop in which the convergence of the adjoint variable `xb` is tested.

The subroutine `SOLVE` is called first and then the solution `x` is passed to the subroutine `SOLVE_B`. Thus there is no need to recompute the solution in the reverse mode.

4.6. Dead adjoint code

Consider the following piece of code from a 2-D finite volume solver. The `do` loop loops over the cell faces, calculates the flux across the face and adds it to the two cells sharing that face.

```
      do ie=1,ne
            call roe(ie, edge, ds, coord, prim, qx, qy, flux)
            do iv=1,nvar
                  divf(iv,edge(1,ie)) = divf(iv,edge(1,ie)) + flux(iv)
                  divf(iv,edge(2,ie)) = divf(iv,edge(2,ie)) - flux(iv)
            enddo
      enddo
```

In the differentiated code listed below, the loop over the cell faces is present but the flux calculated in this loop is not used. Hence this loop constitutes dead adjoint code. It is important to remove such dead code since flux calculation is the costliest part of a finite volume solver.

```
      DO ie=1,ne
        CALL ROE(ie, edge, ds, coord, prim, qx, qy, flux)
      ENDDO
      DO ii1=1,nvar
        fluxb(ii1) = 0.0
      ENDDO
      DO ie=ne,1,-1
        DO iv=nvar,1,-1
          fluxb(iv) = fluxb(iv) + divfb(iv, edge(1, ie)) - divfb(iv,
     +      edge(2, ie))
        ENDDO
        CALL ROE_B(ie, edge, ds, dsb, coord, coordb, prim, primb, qx,
     +             qxb, qy, qyb, flux, fluxb)
      ENDDO
```

```
      program main
      implicit none
      real x, a, cost
      a = 5.0
      call costfunc(x,a,cost)
      print*, 'x  = ', x
      print*, 'a  = ', a
      print*, 'cost = ', cost
      return
      stop
      end

C Cost function
      subroutine costfunc(x,a,cost)
      implicit none
      real x, a, cost
      call solve(x,a)
      cost = x*a
      return
      end

C Iterative solver
      subroutine solve(x,a)
      implicit none
      real x, a, xold, residue
      integer i
c     Starting value for the iterations
      x    = 10.5
100   xold = x
      call iterate(x,a)
      residue = abs(x - xold)
      print*,'x = ',x,' residue = ',residue
      if(residue.gt.0.0001) goto 100
      return
      end

C Update the solution
      subroutine iterate(x,a)
      implicit none
      real x, a
      x = 0.5*x + 0.5*a/x
      return
      end
```

Figure 9: Example of cost function which depends on an iterative solver

```
      SUBROUTINE SOLVE_B(x, xb, a, ab)
      IMPLICIT NONE
      REAL a, ab, x, xb
      INTEGER ad_count, branch, i, i0
      REAL residue, xold

c Forward loop - begin
      x = 10.5
      ad_count = 1
 100  xold = x
      CALL PUSHREAL4(x)
      CALL ITERATE(x, a)
      IF (x - xold .GE. 0.) THEN
        residue = x - xold
        CALL PUSHINTEGER4(1)
      ELSE
        residue = -(x-xold)
        CALL PUSHINTEGER4(0)
      END IF
      IF (residue .GT. 0.0001) THEN
        ad_count = ad_count + 1
        GOTO 100
      END IF
      CALL PUSHINTEGER4(ad_count)
c Forward loop - end

c Reverse loop - begin
      CALL POPINTEGER4(ad_count)
      DO i0=1,ad_count
        CALL POPINTEGER4(branch)
        CALL POPREAL4(x)
        CALL ITERATE_B(x, xb, a, ab)
      ENDDO
c Reverse loop - end
      END
```

Figure 10: Differentiation of code given in figure (9) in reverse mode

```
      SUBROUTINE SOLVE_B(x, xb, a, ab)
      IMPLICIT NONE
      REAL a, ab, x, xb
      REAL residue, xbold

100   xbold = xb
      CALL ITERATE_B(x, xb, a, ab)
      residue = abs(xb − xbold)
      if(residue > 0.0001) goto 100

      END
```

Figure 11: Modification of differentiated code given in figure (10); note that the forward loop has been removed and convergence of adjoint is checked.

4.7. Insensitive variables

There could be several active variables in a code whose dependence on the independent variables is either weak or irrelevant. For example, in a steady state computation, the final solution does not depend on the time-step. But the time-step depends on the current solution (through CFL condition) and an AD tool will differentiate the time-step also. There are two ways to remove this dependency. In the first approach, the time-step can be removed from the primal code before differentiating and added back afterwards. In the second approach, the time-step variable can be passed through a dummy function and TAPENADE will think that the time-step does not depend on any active variables.

5. Pressure matching for quasi-1D nozzle

This problem consists of finding the shape of a quasi-1D nozzle that will generate the given pressure distribution. The flow in the nozzle is governed by the Euler equations for inviscid flow,

$$\frac{\partial U}{\partial t} + \frac{\partial f}{\partial x} = S \tag{20}$$

where

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho E \end{bmatrix}, \quad f = y \begin{bmatrix} \rho u \\ p + \rho u^2 \\ (\rho E + p)u \end{bmatrix}, \quad S = \frac{\mathrm{d}y}{\mathrm{d}x} \begin{bmatrix} 0 \\ p \\ 0 \end{bmatrix}$$

The nozzle shape is given by

$$y(x) = a + b \tanh(cx - d) \tag{21}$$

where

$$a = y_o - b \tanh(-d) \tag{22}$$

where y_o is the radius at the inlet. The desired pressure distribution corresponds to the following values of the design variables,

$$\begin{aligned} b_d &= 0.347 \\ c_d &= 0.800 \\ d_d &= 4.000 \end{aligned}$$

The inlet conditions are $M = 1.5$, $p = 101325$, $\rho = 1$ and a pressure ratio of 2.5 between outlet and inlet. The cost function is defined by

$$I = \int_0^L (p - p_d)^2 \mathrm{d}x \tag{23}$$

The Euler equations in conservation form are solved by a finite volume method with AUSMDV flux. The code is differentiated in direct mode for the cost function with respect to the three design variables using the multi-directional mode of TAPENADE [10]. The differentiated code is modified to check for convergence of the gradients along with the convergence of the flow solution.

5.1. Nozzle test case 1

The initial values for the design variables are

$$\begin{aligned} b &= 0.347 \\ c &= 1.000 \\ d &= 3.800 \end{aligned}$$

The initial value of the cost function is 3.19 and the final value is 1.59e-4 and the design variables are

$$\begin{aligned} b &= 0.35116 \\ c &= 0.76904 \\ d &= 3.85278 \end{aligned}$$

and the gradients are

$$\begin{aligned} \frac{\partial I}{\partial b} &= 5.859 \times 10^{-3} \\ \frac{\partial I}{\partial c} &= -1.490 \times 10^{-3} \\ \frac{\partial I}{\partial d} &= 2.496 \times 10^{-4} \end{aligned}$$

Both the cost function and the gradients have converged by sufficient orders as seen in Figs. (12) and (13) respectively. The convergence of the control variables is shown in Fig. (14). It is seen that convergence is obtained in about 30 iterations. The shape has not converged to the target shape as seen from the final values of the design variables and also from Fig. (15) but the pressure distribution is recovered quite well as seen in Fig. (16).

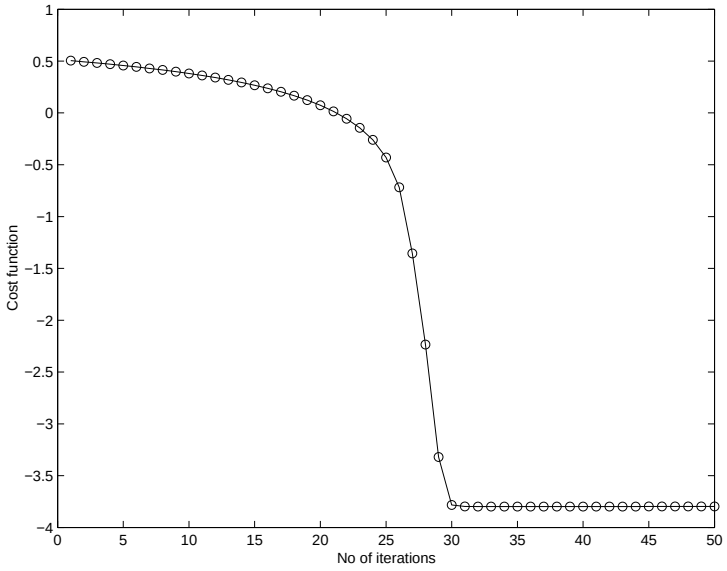


Figure 12: Convergence of cost function for nozzle test case 1

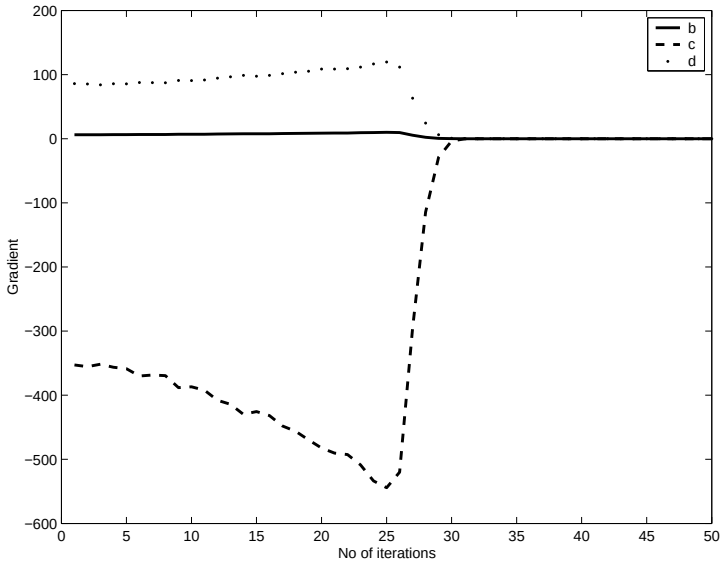


Figure 13: Convergence of gradient for nozzle test case 1

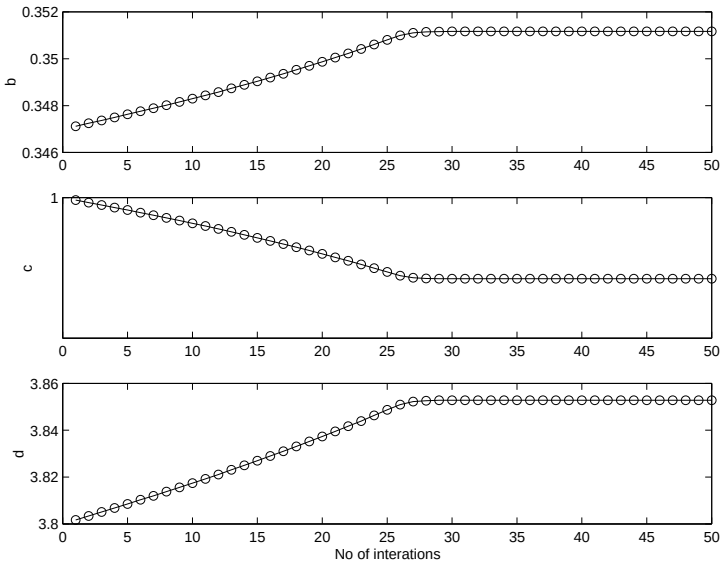


Figure 14: Convergence of control variables for nozzle test case 1

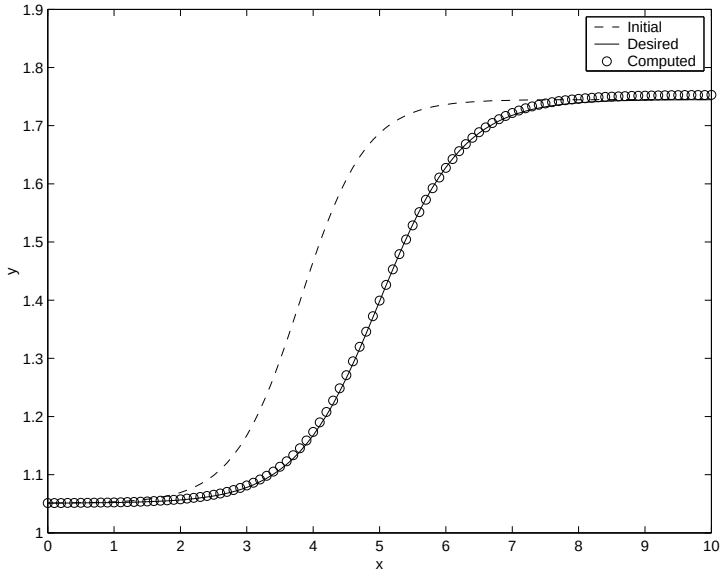


Figure 15: Nozzle shape for nozzle test case 1

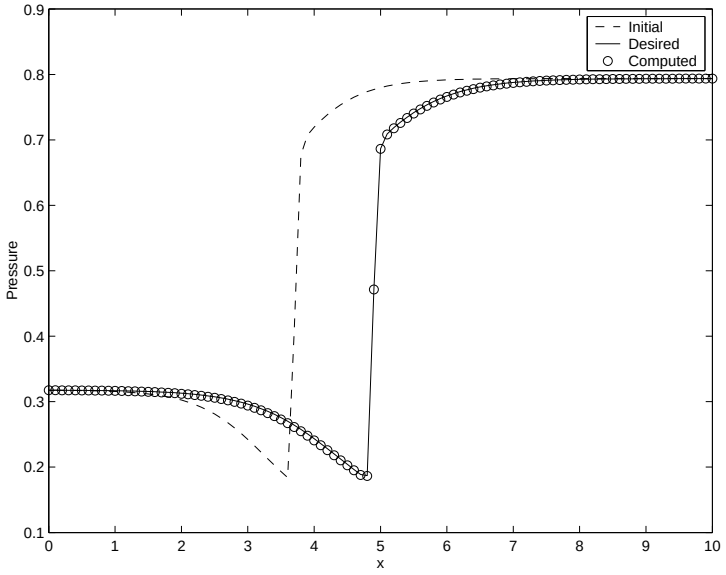


Figure 16: Pressure distribution for nozzle test case 1

5.2. Nozzle test case 2

In this case all the three control variables are perturbed from their design values. The initial values for the design variables are

$$\begin{aligned} b &= 0.32 \\ c &= 1.00 \\ d &= 3.80 \end{aligned}$$

The initial value of the cost function is 3.11 while the final value is 9.21e-5 and the design variables are

$$\begin{aligned} b &= 0.32428 \\ c &= 0.77602 \\ d &= 3.85137 \end{aligned}$$

and the gradients are

$$\begin{aligned} \frac{\partial I}{\partial b} &= 2.949 \times 10^{-3} \\ \frac{\partial I}{\partial c} &= -1.445 \times 10^{-4} \\ \frac{\partial I}{\partial d} &= 3.337 \times 10^{-3} \end{aligned}$$

Both the cost function and gradients have again converged well, see Figs. (17) and (18). The convergence of the control variables is shown in Fig. (19) and we again see that convergence is obtained in about 30 iterations. However the target shape is not recovered as seen from the final values of the design variables and more clearly from Fig. (20). The pressure distribution is however recovered quite well as seen in Fig. (21).

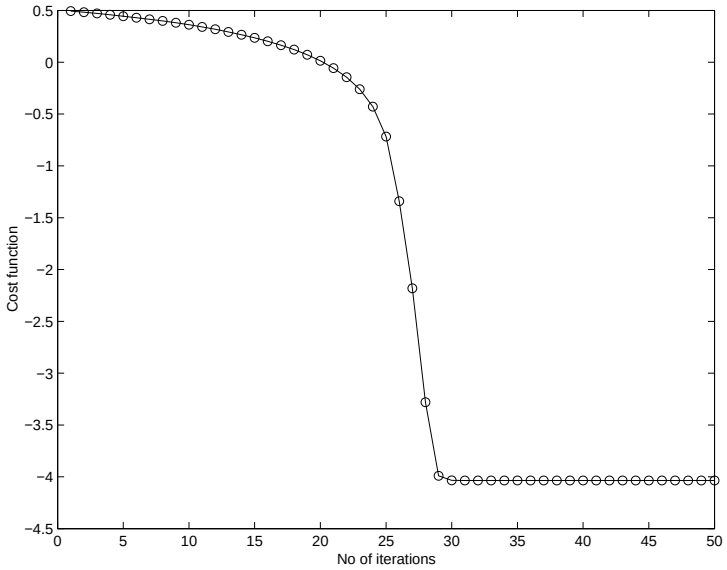


Figure 17: Convergence of cost function for nozzle test case 2

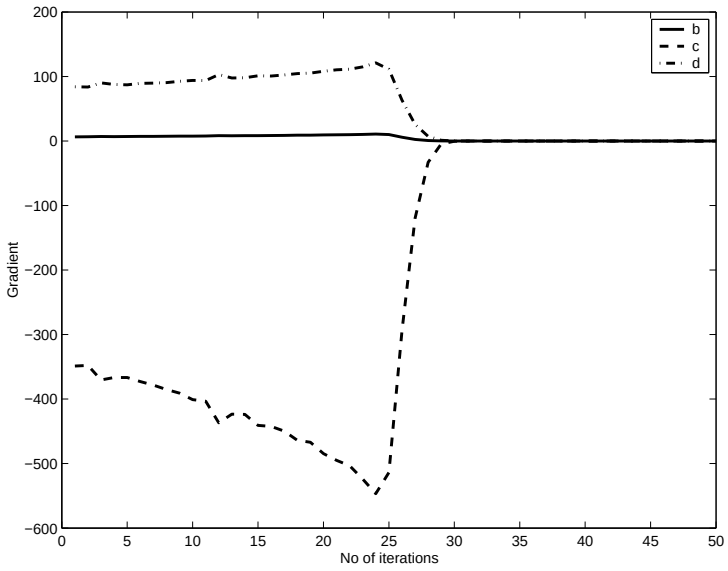


Figure 18: Convergence of gradient for nozzle test case 2

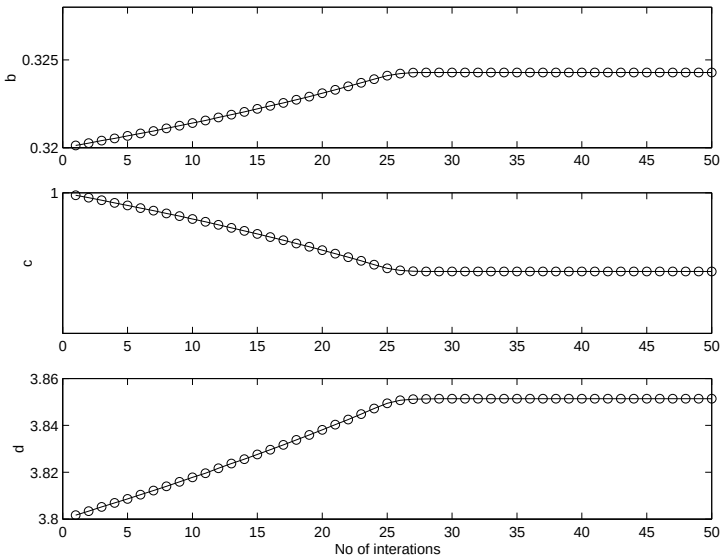


Figure 19: Convergence of control variables for nozzle test case 2

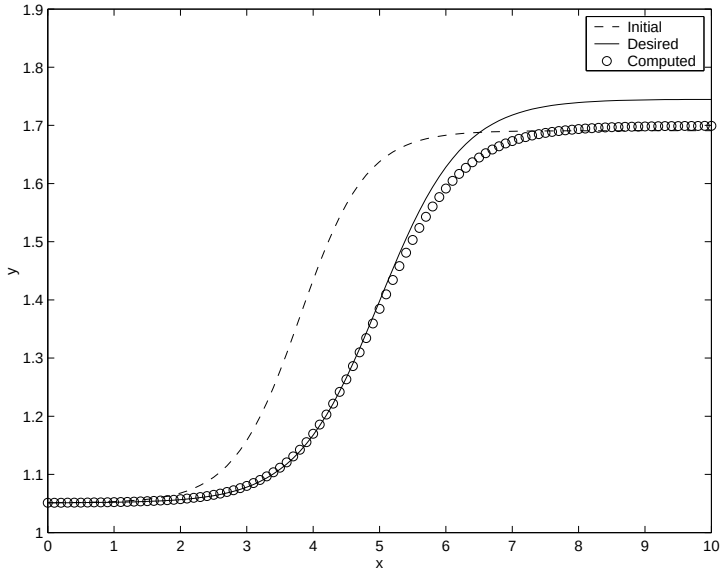


Figure 20: Nozzle shape for nozzle test case 2

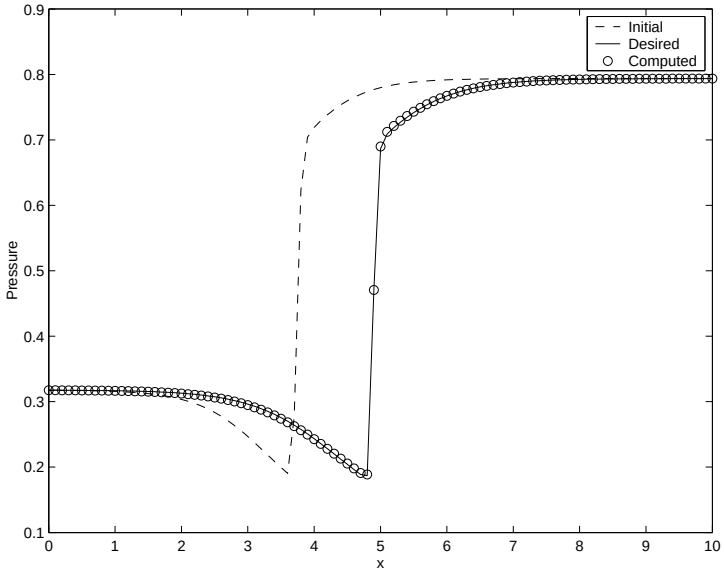


Figure 21: Pressure distribution for nozzle test case 2

6. Control of 1-D Burgers equation

Consider the following Burgers equation with a source term,

$$\begin{gathered} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{u^2}{2}\right) = 0.3xu \\ u(t,-1) = 1, \quad u(t,1) = -0.8, \quad u(0,x) = -0.9x + 0.1 \end{gathered} \tag{24}$$

The steady solution is piecewise parabolic with a jump discontinuity and is given by

$$u(x, t \to \infty) = u_d(x) = \begin{cases} 0.15x^2 + 0.85 & \text{for } x < x_s \\ 0.15x^2 - 0.95 & \text{for } x > x_s \end{cases} \tag{25}$$

where x_s is the shock position given by

$$x_s = -\frac{1}{\sqrt{3}}$$

We consider the following control problem [15]

$$\begin{gathered} \frac{\partial v}{\partial t} + \frac{\partial}{\partial x}\left(\frac{v^2}{2}\right) = F(x)v \\ v(t,-1) = 1, \quad v(t,1) = -0.8 \end{gathered} \tag{26}$$

where the unknown function F is the control variable. The objective is to find the control F which minimizes the following cost function

$$I = \int_{-1}^{+1} (v - u_d)^2 \mathrm{d}x \tag{27}$$

The problem is solved using a finite volume method with 100 cells and Roe flux function. The explicit update equation is

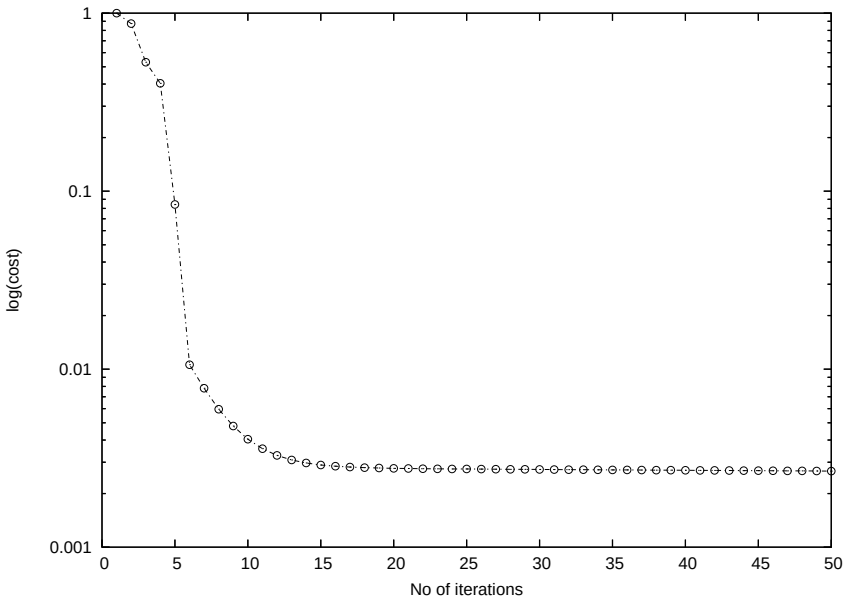


Figure 22: Convergence of cost function

$$v_i^{n+1} = v_i^n - \frac{\Delta t}{\Delta x}(f_{i+1/2}^n - f_{i-1/2}^n) + F_i v_i^n \Delta t$$

The cost function is discretized as

$$I = \sum_{i=1}^{100} [v_i - u_d(x_i)]^2$$

and the discrete values F_i, $i = 1$ to 100 are used as the control variables. Since the number of control variables is large, the reverse mode of AD is used for this problem. The modification strategy outlined in section (4.5) is applied to remove the differentiation of the iterative loop for steady state convergence. The initial values of the control variables are taken to be $F_i = x_i + 0.1\sin(\pi x_i)$ where the second term is the perturbation over the desired control. A constant step-size of 0.002 is used throughout the optimization iterations. Fig. (22) shows the convergence of the cost function which is reduced by about 2.5 orders of magnitude. Fig. (23) shows the gradients after convergence and we notice that the gradients are not sufficiently reduced. Convergence is seen to be achieved in about 15 iterations. Fig. (24) shows the initial, final and the desired control. We see that there is considerable difference between the desired and computed control. The final solution however agrees quite well with the target solution as seen in Fig. (25).

7. Summary and future work

The problem of optimization has been discussed and various methods for computing gradients required for descent methods are introduced. The adjoint approach is valuable when large number of design variables are present as in shape optimization. The discrete adjoint approach

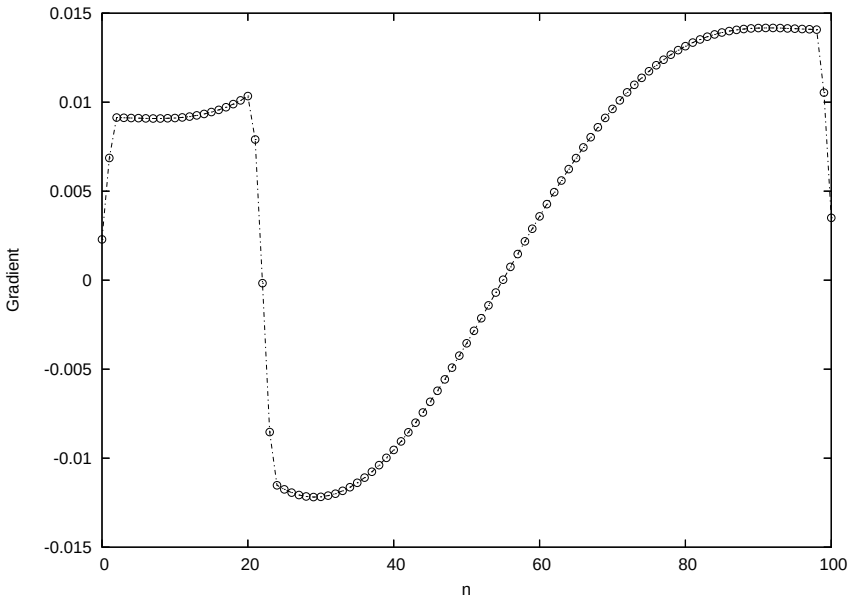


Figure 23: Gradient of cost function

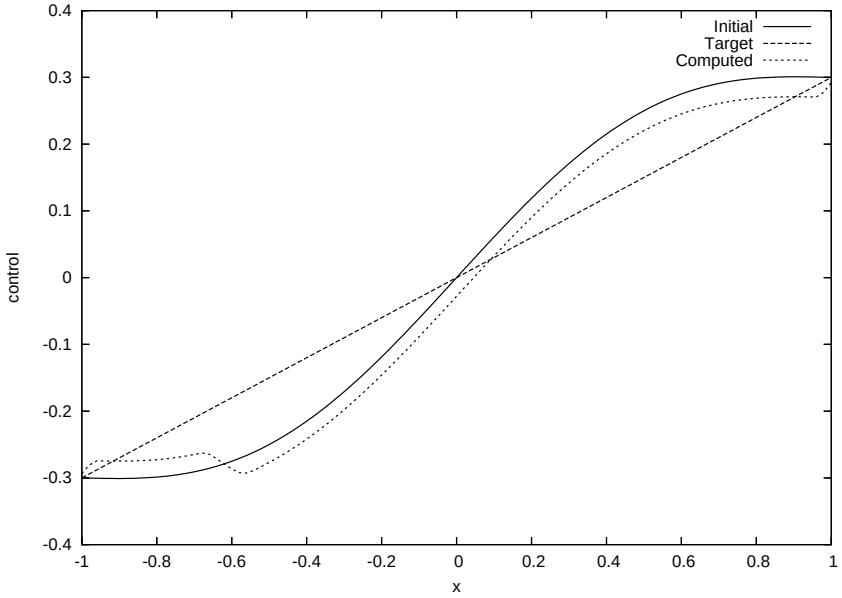


Figure 24: Control function

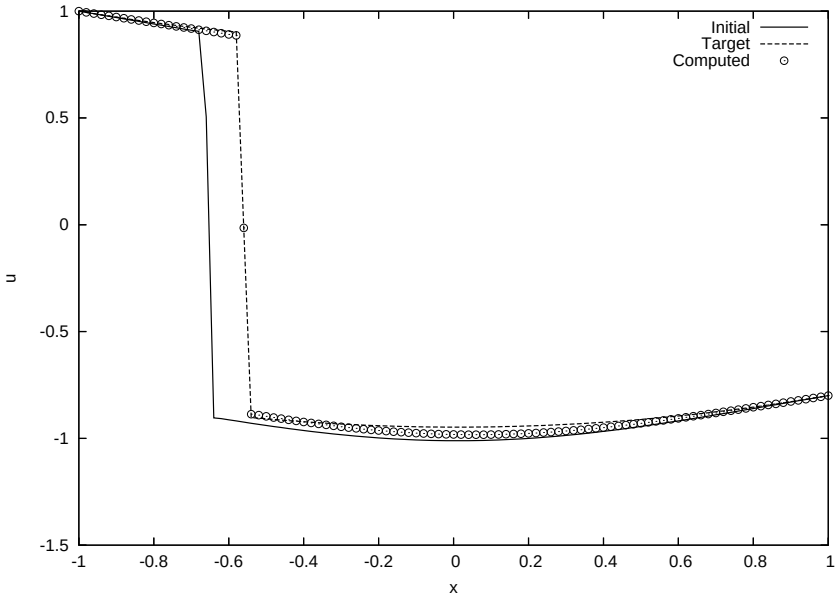


Figure 25: Solution to the control problem

is particularly becoming attractive due to the development of automatic differentiation tools. The efficient use of AD tools requires the state solver like a CFD code to be modified to reduce memory requirements. Some of these issues have been discussed in this report. The presence of iterative loops is particularly difficult to handle since memory requirements are prohibitive. A strategy to modify the differentiated code is suggested and implemented on the Burgers control problem. This is however not an ideal strategy for a complex CFD solver since it requires considerable modification of the code by hand which can introduce errors. Moreover this laborious task would have to be repeated every time the cost function is changed. One alternative is to use the hybrid approach suggested by Courty et al. [6] where the adjoint equation is assembled by hand but all the required jacobians are obtained using an AD tool. A further simplication is to use the approximate gradient idea of Mohammadi in which the dependence of the cost function on the state is ignored. It is then not necessary to solve any adjoint equation and the gradients are obtained very cheaply. Both these techniques will be used in future work involving shape optimization based on Euler and Navier-Stokes equations.

8. Acknowledgements

I would like to thank Dr. Manoj Nair for many useful discussions on optimization and for giving his 1-D nozzle code. I also thank Prof. O. Pironneau and Prof. M. Masmoudi for valuable advice on optimization and AD. Finally, thanks to the TAPENADE team for freely providing their tool and also for answering many questions on their mailing list.

A. A short guide to using TAPENADE

TAPENADE can be invoked at the command line by typing

```
$ tapenade
```

To get help with command line arguments, type

```
$ tapenade -?
```

Lets us illustrate the use of TAPENADE with an example. Let the fortran program file be called `prog.f`, and the independent (input) and dependent (output) variables be called `x` and `y` respectively. We want to differentiate `y` with respect to `x` in direct and reverse modes. Assume that the function or subroutine which computes the values of `y` is called `func`. Then the differentiation is performed by

- Direct mode

  ```
  $ tapenade -d -head func -var x -outvars y prog.f
  ```

- Reverse mode

  ```
  $ tapenade -b -head func -var x -outvars y prog.f
  ```

This will generate subroutines called `func_d` and `func_b` respectively. If any other subroutines are called by the main routine `func` then they are also differentiated since TAPENADE automatically checks for any data dependency. If the subroutines are present in different files then they can be listed on the command line, for example

```
$ tapenade -b -head func -var x -outvars y prog1.f prog2.f prog3.f
```

A more detailed description of the design and use of TAPENADE is given in [11].

References

[1] `http://www.autodiff.org`

[2] W. K. Anderson and V. Venkatakrishnan, "Aerodynamic design optimization on unstructured grids with a continuous adjoint formulation", AIAA Paper No. 97-0643, 1997.

[3] W. K. Anderson and D. L. Bonhaus, "Aerodynamic design on unstructured grids for turbulent flow", NASA Technical Memorandum 112867, June 1997.

[4] 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.

[5] Ciarlet P. G., *Introduction to Numerical Linear Algebra and Optimization*

[6] 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.

[7] J. Elliott and J. Peraire, "Practical 3D aerodynamic design and optimization using unstructured meshes", AIAA Paper No. 96-4170, 1996.

[8] J. Elliott and J. Peraire, "Aerodynamic optimization on unstructured meshes with viscous effects", AIAA Paper No. 97-1849. 1997.

[9] M. B. Giles and N. A. Pierce, "An introduction to the adjoint approach to design", Flow, Turbulence and Combustion, 65(3-4):393-415, 2000.

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

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

[12] A. Jameson, "Optimum aerodynamic design using control theory", *Computational Fluid Dynamics Review 1995*, edited by M. Hafez and K. Oshima, pp. 495-528, Wiley, 1995.

[13] A. Jameson and Sriram, "A continuous adjoint method for unstructured grids", AIAA 2003-3955, 2003.

[14] A. Jameson, Sriram, K. Martinelli and B. Haimes, "Aerodynamic shape optimization of complete aircraft configurations using unstructured grids", AIAA 2004-0533.

[15] Mohammadi B. and Pironneau O., *Applied Shape Optimization for Fluids*, Clarendon Press, Oxford, 2001.

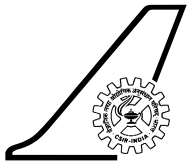
[16] Manoj T. Nair., "Development of a two-dimensional aerodynamic optimisation code based on sensitivity analysis", Project Document CF-0508, National Aerospace Laboratories, Bangalore, June 2005.

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

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

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

Documentation Sheet



National Aerospace Laboratories

Class Unrestricted

No. of Copies 10

Title	Using automatic differentiation for solving optimization problems
Author(s)	Praveen. C

Division	CTFD	**Project No.**	C-8-117
Document No.	PD CF 0514	**Date of Issue**	December, 2005

Contents	36 Page(s)	28 Figure(s)	0 Table(s)	19 Reference(s)

External Participation None

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

Abstract

Gradient-based optimization is useful for designing engineering systems and for improving existing designs. The efficient computation of gradients for systems governed by partial differential equations is non-trivial. Different approaches to compute gradients are discussed including adjoint equations and automatic differentiation.

An automatic differentiation tool called TAPENADE is used to compute sensitivities for optimization. The tool supports both direct and reverse modes of differentiation. In the reverse mode, a compute-store strategy is used which leads to large storage requirements. This can however be minimized by making small changes to the primal code. The tool is first applied to some test codes to illustrate the idea of automatic differentiation. Two optimization problems are solved. The first problem involves a quasi-one dimensional flow in a nozzle and the problem of pressure matching is solved using the direct mode. The second problem involves a control problem for the 1-D Burgers equations which is solved in the reverse mode. Both the problems illustrate that inverse problems can have non-unique solutions.