Automatic Differentiation and Navier-Stokes Computations

Paul Hovland, Bijan Mohammadi, and Christian Bischof

CRPC-TR97768-S
October 1997

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Submitted September 1998; Also available as Argonne Techreport ANL/MCS-P687-0997

Automatic Differentiation and Navier-Stokes Computations

Paul Hovland* Bijan Mohammadi† Christian Bischof*

We describe the use of automatic differentiation (AD) in, and its application to, a compressible Navier-Stokes model. Within the solver, AD is used to accelerate convergence by more than an order of magnitude. Outside the solver, AD is used to compute the derivatives needed for optimization. We emphasize the potential for performance gains if the programmer does not treat AD as a black box, but instead utilizes high-level knowledge about the nature of the application.

1. Introduction

The Navier-Stokes equations are frequently used to model the flow of compressible or incompressible fluids. Automatic differentiation (AD) is an emerging technology for differentiating functions that enables derivatives to be computed accurately and efficiently, with very little effort on the part of the programmer. In this paper, we discuss how AD can be used to enhance a compressible Navier-Stokes solver. Section 2 describes the two-dimensional Navier-Stokes model and solver used in our studies. Section 3 gives a brief introduction to source transformation tools for automatic differentiation. Section 4 discusses how derivatives computed using AD can be used for shape optimization. Section 5 explains how an explicit solver can be transformed into an implicit solver using a Jacobian computed using AD. Section 6 briefly describes how AD might be used in optimal control. We conclude with a summary of our results and a discussion of how insight into the high-level mathematics of a computation can greatly reduce the cost of derivative computations using AD.

2. The Navier-Stokes Solver

The time-dependent Navier-Stokes equation can be written as

$$\frac{\partial U}{\partial t} + \nabla \cdot [F(U) - N(U)] = 0,$$

where U is the vector of state variables and F and N are the convective and diffusive operators. When the Reynolds and/or Raleigh numbers are high, turbulence models are necessary. Adding the $k-\varepsilon$ turbulence model to the

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439-4844, email: {hovland,bischof}@mcs.anl.gov.

†University of Montpellier II and INRIA, Math. Dept, CC51, 34095 Montpellier Cedex 5, France, email: Bijan.Mohammadi@inria.fr.

equation yields

$$\frac{\partial U}{\partial t} + \nabla \cdot [F(U) - N(U)] = S(U),$$

where $U = (\rho, \rho u, \rho v, \rho E, \rho k, \rho\varepsilon)^t$ and S contains the source terms of the k–ε model. In two dimensions, this provides six equations in seven variables. We close the system using the equation of state $p = p(\rho, T)$. Note that this model does not take into account turbulent contributions to the pressure and total energy.

In this paper, we consider the numerical solution of this model using the NSC2KE program [14]. The numerical solution is based on a finite volume-Galerkin method for unstructured triangular meshes. The convective part of the equations can be solved using either a Roe [17] or Osher [16] approximate Riemann solver. Second order accuracy is obtained by using a MUSCL-like extension with Van Albada type limiters [1]. Applying this approach to the $k-\varepsilon$ equations will not guarantee the positivity of ρk and $\rho\varepsilon$. Therefore, the convective fluxes for k and ε are computed using the PSI fluctuation splitting scheme [20]. The viscous term is discretized using a classical piecewise linear finite element method. Time discretization is based on a four-stage Runge-Kutta scheme. See [14, 15] for more details.

3. Automatic Differentiation via Source Transformation

Automatic differentiation (AD) works by exploiting the fact that all algorithms compute functions through the composition of elementary functions provided by the programming language. Partial derivatives for these elementary functions can be obtained by table lookup, and total derivatives propagated using the chain rule. A thorough description of AD is beyond the scope of this paper. In the following paragraphs, we briefly discuss one method for implementing AD; see [8] for an explanation of the theoretical foundations of AD and [12] for a survey of other methods.

Source transformation AD tools work by converting source code for computing a function into source code for computing that function and its derivatives. For example, the ADIFOR tool [5] for AD of Fortran 77 converts the subroutine in Figure 1 into the subroutine in Figure 2. The *derivative objects* `g_x`, `g_temp` and `g_y` contain the derivatives of `x`, `temp`, and `y` with respect to `x`. Typically, on entry `g_x` will contain the $n \times n$ identity matrix and on exit `g_y` will contain the Jacobian matrix dy/dx. Similarly, the Odyssée tool [18] converts the same subroutine into the one in Figure 3. Variables `yccl` and `xccl` contain the derivatives of `y` with respect to `y` and `x`. The `save*` variables are used to save intermediate values. Typically, on entry `yccl` will contain the $m \times m$ identity matrix and on exit `xccl` will contain the Jacobian matrix dy/dx. This example also illustrates the difference between these tools. The ADIFOR tool uses the

```
subroutine mv_prod(a,x,y,m,n)

integer m,n,i,j
real a(m,n),x(n),y(m),temp

do i=1,m
   y(i) = 0.0
enddo
do i=1,n
   temp = x(i)
   do j=1,m
      y(j) = y(j) + a(j,i)*temp
   enddo
enddo

return
end
```

Figure 1: Code for computing a simple function (y = Ax).

forward mode of AD, propagating derivatives from the independent variable x to the dependent variable y. The code produced by the Odyssée tool uses the *reverse mode* and propagates derivatives from the dependent variable y to the independent variable x. Note that because the derivative computation of the reverse mode reverses the data flow of the function computation, there is a need to save all intermediate values. In general, the reverse mode has much greater memory requirements than the forward mode, but can offer better performance when the number of scalar independent variables is significantly greater than the number of scalar dependent variables. For our example, this occurs when $m << n$. See [8, 9] for more details.

4. Shape Optimization

Simulation of the flow around a shape defined by certain shape parameters x_s typically follows the computational path

$$x_s \longrightarrow x_w \longrightarrow x_m \longrightarrow U,$$

where x_w is the discretized surface, x_m is the set of mesh nodes, and U is the computed flow. For shape optimization, we try to minimize a cost function $J(U, x_w)$, which may involve constraints on the geometry ($g_1(x_w) \leq 0$) and

```
C     [Disclaimer deleted]
C
      subroutine g_mv_prod(g_p_,a,x,g_x,ldg_x,y,g_y,ldg_y,m,n)
C
      integer m, n, i, j
      real a(m, n), x(n), y(m), temp
C
      integer g_pmax_
      parameter (g_pmax_ = 100)
      integer g_i_, g_p_, ldg_y, ldg_x
      real g_y(ldg_y, m), g_temp(g_pmax_), g_x(ldg_x, n)
      save g_temp

      if (g_p_ .gt. g_pmax_) then
        print *, 'Parameter g_p_ is greater than g_pmax_'
        stop
      endif
      do i = 1, m
        do g_i_ = 1, g_p_
          g_y(g_i_, i) = 0.0
        enddo
        y(i) = 0.0
C------
      enddo
      do i = 1, n
        do g_i_ = 1, g_p_
          g_temp(g_i_) = g_x(g_i_, i)
        enddo
        temp = x(i)
C------
        do j = 1, m
          do g_i_ = 1, g_p_
            g_y(g_i_, j) = g_y(g_i_, j) + a(j, i) * g_temp(g_i_)
          enddo
          y(j) = y(j) + a(j, i) * temp
C------
        enddo
      enddo
C
      return
      end
```

Figure 2: Code for computing a function and its Jacobian, generated by ADIFOR.

```
      subroutine mv_prodcl (a, x, y, m, n, xccl, yccl)

C [ODYSSEE output edited for brevity]

      integer m, n, i, j
      real a(m,n), x(n), y(m), temp
      integer odyn,odym
      parameter (odyn = 10, odym = 10)
      dimension yccl(m), xccl(n), save3(1:odym, 1:odyn)
      dimension save2(1:odyn), save1(1:odym)

      tempccl = 0.

      do i = 1, m
         save1(i) = y(i)
         y(i) = 0.0
      end do
      do i = 1, n
         save2(i) = temp
         temp = x(i)
         do j = 1, m
            save3(j,i) = y(j)
            y(j) = y(j)+a(j,i)*temp
         end do
      end do

      do i = n, 1, -1
         do j = m, 1, -1
            y(j) = save3(j,i)
            tempccl = tempccl+yccl(j)*a(j,i)
         end do
         temp = save2(i)
         xccl(i) = xccl(i)+tempccl
         tempccl = 0.
      end do
      tempccl = 0.
      do i = m, 1, -1
         y(i) = save1(i)
         yccl(i) = 0.
      end do
      return
      end
```

Figure 3: Code for computing a function and its Jacobian, generated by Odyssée.

the flow ($g_2(U) \leq 0$). This minimization problem can actually be viewed in two ways:

1. Find x_s to minimize $J(U(x_s), x_w(x_s))$.
2. Find x_w to minimize $J(U(x_w), x_w)$.

The first approach is the one more commonly used, in part because the number of shape parameters x_c is usually smaller than the number of surface discretization points x_w and also because it is easier to ensure that the surface remains smooth. However, this approach suffers from the following limitations.

1. Computing $x_w(x_c)$ is hard, especially in three dimensions. Usually, this is done by a "black box" CAD tool. We can't compute $\frac{\partial x_w}{\partial x_c}$ analytically, which limits our ability to use a gradient-based optimization method.
2. Parameterized shapes may place unnecessary restrictions on the solution space X_w.

We describe a technique that uses AD and mesh deformation to perform a CAD-free shape optimization using steepest descent.

4.1. Computing the Gradient

We have a number of options in computing the gradient $\frac{dJ}{dx_w}$. We could approximate it using finite differences, but this can introduce truncation error and furthermore is very expensive when there are many surface discretization points, as is usually the case. We could also implement the adjoint code by hand, but this can be extremely tedious and hence error-prone. Instead, we use the reverse mode of automatic differentiation to do this. For our problem, we used the Odyssée automatic differentiation tool [18].

4.2. Adjusting the Shape

Once we have computed $\frac{dJ}{dx_w}$, we use steepest descent to compute a perturbation to x_w, δx_w. To avoid oscillations, a "local" smoothing operator is defined over the shape. This can be, for instance, a few "local" Jacobi iterations to solve the following system:

$$(I - \varepsilon\Delta)\delta\tilde{x}_w = \delta x_w, \tag{1}$$

$$\delta\tilde{x}_w = \delta x_w = 0 \quad \text{on wedges},$$

where $\delta\tilde{x}_w$ is the smoothed shape variation for the shape nodes and δx_w is the variation given by the optimization tool. By "local" we mean that if the predicted shape is locally smooth, it remains unchanged during this step. The regions where the smoothing is applied are identified using a discontinuity-capturing operator. More precisely, ε is set to zero (this is done during the Jacobi loops) if

$$\frac{\delta_{ij}(\delta x_w)}{(\delta x_w)_T} < 10^{-3},$$

where $\delta_{ij}(\delta x_w)$ is the difference between the variations of the nodes of a boundary edge and $(\delta x_w)_T$ is the mean variation on this edge.

The following argument illustrates the importance of this step:

We want the variation $\delta x_w \in C^1(\Gamma)$, if Γ defines a manifold of dimension $(n-1)$ in a domain $\Omega \in R^n$. From Sobolev inclusions, we know that $H^n(\Gamma) \subset C^1(\Gamma)$. It is easy to understand that the gradient method will not necessarily produce $C^1(\Gamma)$ variations δx_w and therefore we need to project them into $H^n(\Gamma)$ (an example of this is given in Figure 4.2). This means that the projected variation $(\delta\tilde{x}_w)$ is the solution of an elliptic system of degree n. However, if we are using a P^1 discretization, a second order elliptic system is sufficient because the wedges of the geometry are considered as constraints for the design. Therefore we project the variations only into $H^2(\Gamma)$ even in 3D.

One advantage of this approach over splines is that singular points (or wedges) may appear if necessary.

4.3. Mesh deformation

Once δx_w has been determined, we need to extend these variations over all the mesh. This is done by solving a volumic elasticity system of the same form as Equation 1:

$$(I - \varepsilon\Delta)\delta x_m = \overline{\delta x_m}, \tag{2}$$

$$\delta x_m = 0 \quad \text{on inflow and outflow boundaries,}$$

$$\frac{\partial \delta x_m}{\partial n} = 0 \quad \text{on slipping boundaries,}$$

$$\delta x_m = \overline{\delta x_m} \quad \text{on wall nodes,}$$

where $\overline{\delta x_m}$ is the extension of δx_w over the mesh defined by

$$\overline{\delta x_m} = \delta x_w \quad \text{for mesh nodes on the wall,}$$

$$\overline{\delta x_m} = 0 \quad \text{for internal nodes.}$$

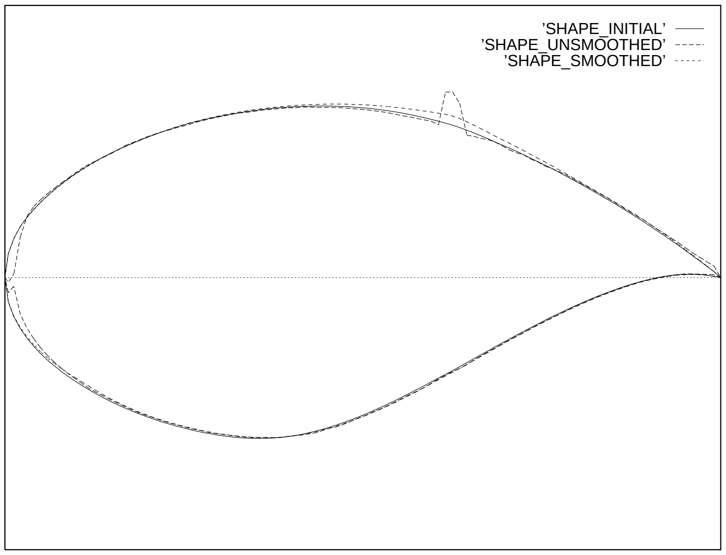

Figure 4: Initial, smoothed and unsmoothed shapes for a transonic drag reduction problem. Note that the gradient jumps through shocks and also produces a non-smooth shape in leading edge regions after one iteration of the optimization. Subsequent iterations increase the non-smoothness.

4.4. Improving Computational Efficiency

The method discussed thus far is inadequate for solving realistic problems. The reason stems from the tradeoff between storage and time complexity present in all adjoint methods. The reverse mode of AD is at one extreme, achieving a time complexity bounded by a constant multiple of that for the function evaluation, at the cost of a potentially exponential increase in storage complexity [8]. Griewank has demonstrated that it is possible to achieve logarithmic growth in time and space through checkpointing and recomputation of intermediate values [9].

For a large problem that takes many time iterations to reach a steady state solution, the storage requirements for the basic reverse mode can be enormous. The flow solver has the following structure.

```
time = 0
repeat
  foreach triangle
    update triangle
  end
  foreach edge
    update edge
  end
  foreach node
    update node
  end
  time = time + deltat
until steady state
```

Each update typically involves on the order of 10^2 intermediate values. Thus, if we have k time steps and N triangles, we have on the order of $300kN$ intermediate values that must be stored. For a moderately sized system involving 10^4 triangles and 10^3 timesteps, the storage requirements can exceed 10 gigabytes. Fortunately, by using a little insight into the mathematics of the problem and the structure of the computation, we can dramatically reduce the memory requirements.

The first optimization takes advantage of the fact that our cost function depends only on the flow at the steady state. If our convergence criterion is sufficiently strict [10, 15], it is sufficient to only differentiate through one time step, if we start with an initial state corresponding to the steady state for a given shape. This reduces the memory requirements by a factor k (the number of time steps). This technique would not apply in the case of unsteady flows.

The second optimization relies on a technique called *inter-procedural differentiation* [7]. This method can be viewed as a heuristic for checkpointing, and relies on the way in which Odyssée and other source transformation tools implement the reverse mode for subroutines. A differentiated version of each subroutine is created by adding adjoint computation at the end of the function computation, after augmenting the function computation so that all intermediate variables are saved. Calls to subroutines are replaced in the adjoint phase by calls to the differentiated version of the subroutine. Thus,

```
subroutine A          compute phase I
    call subroutine B
    compute phase II
return
```

becomes

```
differentiated version of subroutine A
    compute phase I (save intermediate values)
    call subroutine B
    compute phase II (save intermediate values)
    adjoints for phase II
    call differentiated version of subroutine B
    adjoints for phase I
return
```

Note that the intermediate values computed in subroutine `B` are not saved. Instead, they are *recomputed* by the differentiated version of subroutine `B`. This provides the foundation for the interprocedural differentiation technique. If we use a subroutine to encapsulate the computation in a loop, the differentiated version will recompute the intermediate values of the loop body, rather than storing them. This only doubles the time requirements for the differentiated loop, but can reduce the memory requirements by a factor equal to the number of intermediate values in the loop body. By applying this technique to the loops over triangles, edges, and nodes, we reduce the memory requirements by another factor of 100.

4.5. Experimental Results

In this section we present some results of constrained optimization problems at various Mach numbers. All computations have been performed on a workstation capable of about 10 Megaflops with approximately 64 Megabytes of memory. The mesh adaptions, gradient computations and flow solutions require less than 2 hours. Using the optimization described in the previous section, the reverse mode requires approximately 10 times more memory than the direct solver. An estimation of the memory required by the direct solver is $50N$ words, where N is the number of nodes. This is more than what is necessary in a structured solver because the data structures involved are much more complicated in an unstructured approach.

In the following examples, when global constraints are present, the different penalty coefficients in the cost function are initially chosen for the different quantities involved to have variations of the same order of magnitude. During optimization, they are reduced with the same ratio as the cost function.

The same drag reduction problem with constraints on the lift and the volume has been considered for various airfoils, wing and a full aircraft. The cost function for all these cases is given by:

$$J(x) = C_d + \alpha |C_l - C_l^0| + \beta |Vol - Vol_0|,$$

where α and β are penalty parameters, C_d is the drag coefficient, C_l and C_l^0 are the actual and initial lift coefficients and Vol and Vol_0 the actual and initial volumes. The adaptive optimization algorithm has been used. To guarantee that the solutions are mesh-independent, a final computation has been done on the final shape until convergence. For these cases, we show the initial and final meshes and the iso-Mach distributions.

4.5.1. Drag reduction for a Naca 0012

The initial airfoil is the NACA 0012. The design takes place at Mach number 0.754 and 2 degrees of incidence. The drag coefficient has been reduced by more than 10 percent while the lift and volume slightly increased.

4.5.2. Drag reduction for a supersonic flow

Our aim here is to show that this approach does not suffer from a change in the nature of the equations when passing from transonic to supersonic regime. The design takes place at Mach number of 2. Again, the initial profile is a NACA0012. For the shock to be attached, the leading edge has to be sharp. But the initial shape has a smooth leading edge. This means that the optimization procedure has to be able to treat the apparition of singular points. As we have noted, this cannot be done if we use splines for instance. No particular treatment has been done for this case.

The drag has been reduced by about 20 percent (from 0.09 to 0.072) as the shock is now attached while the volume has been conserved (from 0.087 to 0.086). The initial lift coefficient should be zero as the airfoil is symmetric. Due to numerical errors, the lift varies from -0.001 to 0.0008. This means that the final shape is almost symmetric.

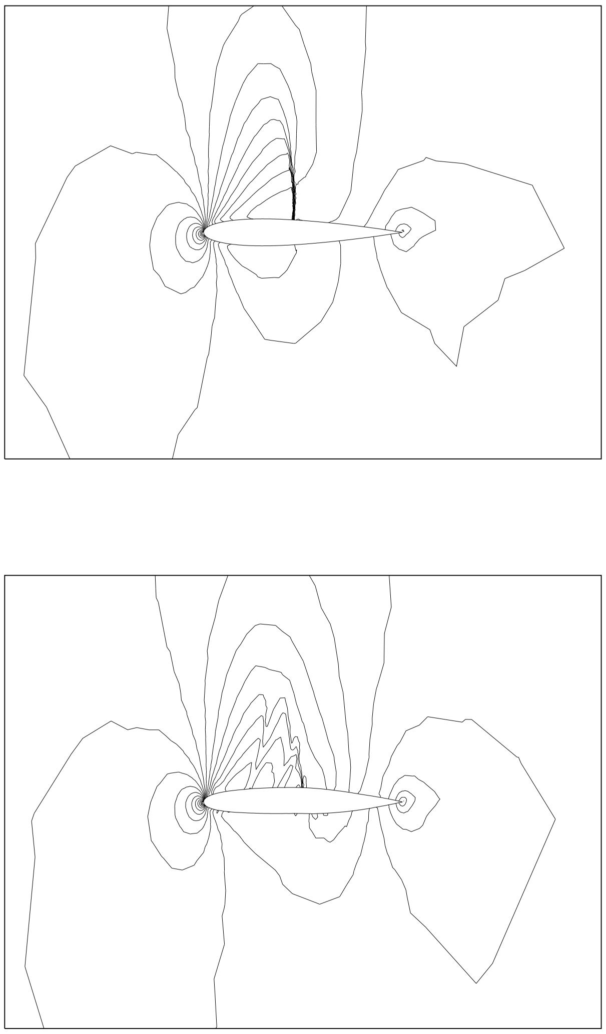

Figure 5: Transonic drag reduction: iso-Mach curves for the initial and final (after 20 iterations) designs.

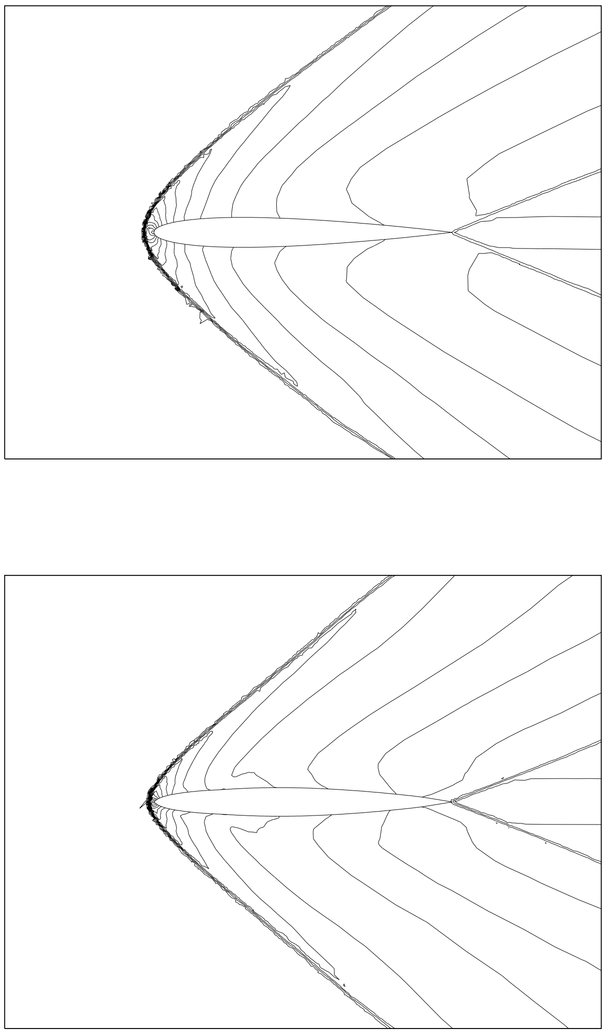

Figure 6: Supersonic drag reduction: iso-Mach curves for the initial and final (after 11 iterations) designs.

5. Switching to an Implicit Solver

For many problems, the time scale of the physical system being simulated is so large that explicit methods converge very slowly, if at all. Therefore, implicit time integration becomes necessary. Developing an implicit solver for a nonlinear PDE often demands a great deal of additional work beyond the construction of an explicit solver.

One way to develop an implicit solver for a nonlinear PDE is to linearize the system using a first-order Taylor expansion. After discretization, many nonlinear PDEs can be rewritten using the general form

$$\frac{\Delta u}{\Delta t} + \Psi(u) = 0,$$

where u is the discretized state and $\Psi(u)$ is a nonlinear function of u. One can then linearize $\Psi(u)$ as

$$\Psi(u^n + \Delta u) \approx \Psi(u_n) + \Psi_u(u_n)\Delta u.$$

For a complicated function $\Psi(u)$ such as the one used in NSC2KE, developing by hand the code to compute the Jacobian matrix $\Psi_u(u)$ can be extremely time-consuming.

However, using AD, it is a relatively simple task to convert an explicit solver to an implicit one. Since the computation of $\Psi(u)$ has already been implemented for use in the explicit method, we can apply AD to this code and automatically generate code for computing the Jacobian $\Psi_u(u)$. Then, given iterate u^n and a timestep Δt, we can find u^{n+1} by solving the linear system

$$\left(\frac{1}{\Delta t} I + \Psi_u(u^n)\right) \Delta u = -\Psi(u^n) \tag{3}$$

and letting $u^{n+1} = u^n + \Delta u$. We can solve (3) using GMRES [19] with ILU preconditioning. We used the linear equation solver provided by the PETSc toolkit for scientific computing [4]. Note that without preconditioning, the implicit solver may converge very slowly, if at all.

5.1. Improving Computational Efficiency

We have thus far demonstrated that using readily available tools for scientific computing, it is a simple task to transform an explicit method into an implicit method based on the linearized system. However, there is room for improvement in the implicit method. The computation of the Jacobian can be accelerated by employing the technique of Averick et al [2] to compute a compressed Jacobian using dense derivative vectors. An additional small improvement can be achieved through the use of the interface contraction technique described in [11]. Finally, the nonlinearity of the function $\Psi(u)$

can be handled better if we solve the system of nonlinear equations directly rather than first linearizing the system.

5.1.1. Compressed Jacobian

Since the Jacobian being computed is very sparse, we use the SparsLinC library [6], which provides support for sparse derivative vectors. SparsLinC offers considerably better performance than would a dense Jacobian computation. However, because of the overhead associated with manipulating the sparse derivative vectors, the cost is considerably more than it would be for a computation involving dense vectors of the same length. In [2], Averick et al describe a technique for compressing the Jacobian based on a coloring algorithm that identifies structurally orthogonal columns of the Jacobian. We adapted this technique to our method.

The first time the Jacobian is computed, the sparsity structure is not known, so we use SparsLinC to compute a sparse Jacobian. Because the mesh does not change, subsequent iterations have the same sparsity pattern. By coloring the Jacobian matrix from the first iteration, we are able to compute a compressed Jacobian for subsequent iterations.

5.1.2. Interface Contraction

In [11], Hovland et al describe a heuristic for reducing the cost of computing derivatives based on the observation that the number of parameters passed to a subroutine is usually quite small. If the number of such variables is small relative to the number of independent variables, it is referred to as an interface contraction. Our model did not exhibit this property, because all variables were passed between subroutines using common blocks. However, we were able to easily isolate a main computational kernel within a subroutine with a relatively small number of parameters. We then applied the interface contraction technique to this subroutine.

5.1.3. Reformulating as a system of nonlinear equations

The implicit method described thus far relies on a linearization of the function $\Psi(u)$ as $\Psi(u) \approx \Psi(u^n) + \Psi_u(u^n)(u - u_n)$ together with the use of a linear equation solver. We can instead use a nonlinear equation solver to solve the nonlinear system

$$\frac{u^{n+1} - u^n}{\Delta t} + \Psi(u^{n+1}) = 0$$

directly. PETSc provides nonlinear equation solvers based on linesearch and trust region methods. We used PETSc to implement an inexact Newton method with linesearching, using GMRES with ILU preconditioning for the linear system solve and a cubic linesearch.

5.2. Experimental results

We applied automatic differentiation to the function $\Psi(u)$ computed by NSC2KE, the 2-dimensional compressible Navier-Stokes solver described earlier. This function is implemented using approximately 1500 lines of Fortran. We used ADIFOR version 2.0 with the SparsLinC library. We used the linear and nonlinear equation solvers provided by PETSc. We then computed the flow in a room with isothermal walls and gravity, but without turbulence.

Figure 7 illustrates that the steady state solutions found by the implicit and explicit solvers are virtually identical. Figure 8 shows the flow computed. Figure 9 demonstrates the necessity for preconditioning, and the superior performance of the implicit method compared to the explicit method. All timings were performed on a Hewlett Packard K9000 with standard (level 2) optimization.

Figure 10 shows the effects of the improvements described in Section 5.1. For the problem considered, the time to find a solution was reduced from 371 cpu seconds for the basic implicit method to 116 cpu seconds for the method based on the nonlinear formulation. Most of this improvement is due to the use of compressed Jacobians. The compressed Jacobian computation is approximately 7 times faster than the sparse Jacobian computation, reducing the overall solution time by almost a factor of 3. The use of interface contraction reduces the cost of the compressed Jacobian computation by about 15%, which results in a reduction of about 5% in the overall solution time. This is only a modest improvement, and indeed was less than our original expectations, but it also required only a few hours of effort. Finally, switching to a true nonlinear equation solver reduces the overall solution time by about 20%.

6. Future Work

We are currently using AD on the outside of NSC2KE for use in an optimal control problem. In this scenario, we try to optimize the lift and drag properties of a 4-element airfoil by rotating the leading element. We use a smooth mesh transformation to prevent numerical variation due to re-meshing. Preliminary results indicate that the optimization of the control parameter converges faster and with greater accuracy using derivatives computed using AD rather than finite difference approximations.

We hope to speed up the flow solver used in this problem by switching to the implicit version described in Section 5. In order to compute the derivative of the cost function J with respect to the angles of rotation α, we will need to differentiate through this solver. This will provide further demonstration of the usefulness of AD, as well as another opportunity for taking advantage of high-level mathematical insight into a problem. Rather

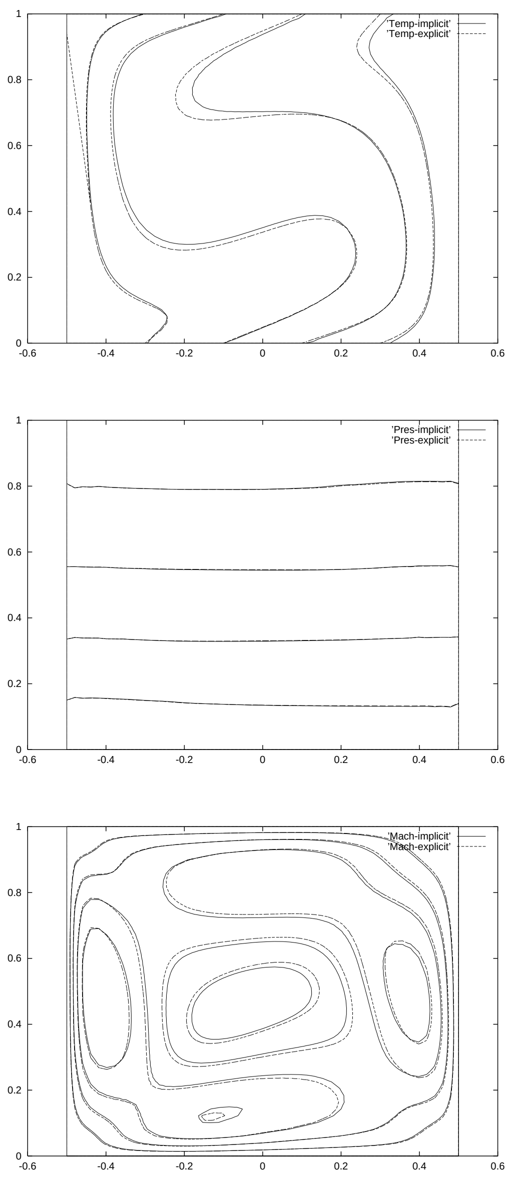

Figure 7: The steady state solutions found by the explicit (solid lines) and explicit (dashed lines) solvers are virtually identical. Top to bottom are iso-curves for temperature, pressure, and mach.

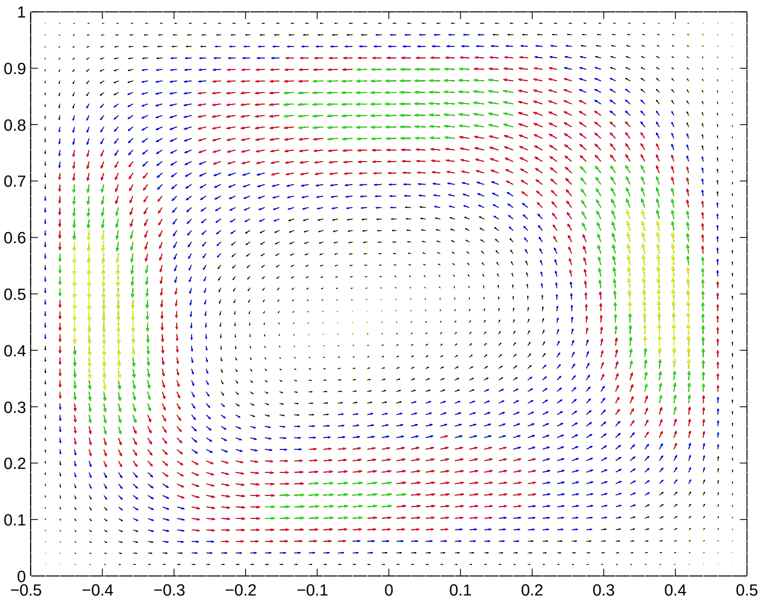

Figure 8: The flow computed by the implicit solver.

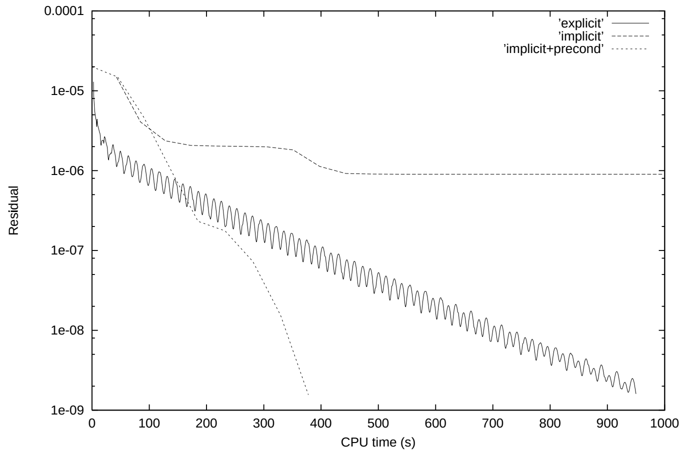

Figure 9: An implicit method based on GMRES with ILU preconditioning offers performance superior to that of an explicit Runge-Kutta method and the implicit method without preconditioning.

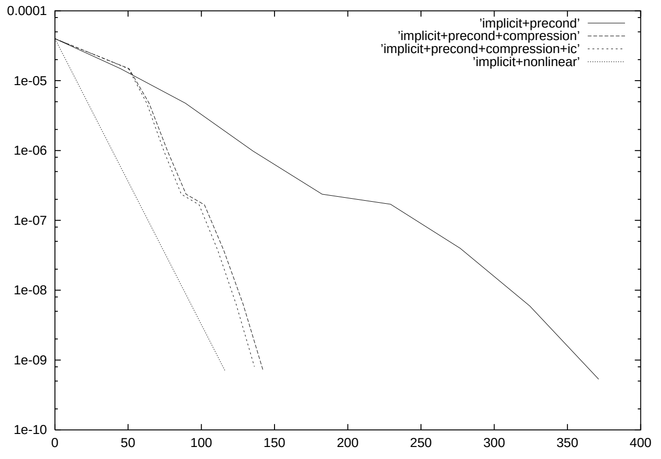

Figure 10: Use of coloring to compute a compressed Jacobian and reformulation as a nonlinear problem offers further improvements in performance.

than differentiating through the PETSc nonlinear equation solver, we can first compute the steady state solution $u^{\infty}(\alpha)$, then use the incremental iterative method described by Sherman et al [13] (See also [3]) to compute the derivative $d\,u/d\,\alpha$.

7. Conclusions

AD tools can enhance Navier-Stokes calculations in a variety of ways. They can be used to generate code for computing the Jacobian matrix used in an implicit solver. They can also be used to generate sensitivities of the flow (or a cost function based on this flow) with respect to the input parameters, including shape and physical parameters. These sensitivities can be used in optimal design or control problems, or they can be used to construct a tangent linear model of the behavior of the system when a parameter such as angle of attack is altered. No matter what purpose AD tools are used for, there is a benefit to be gained from a high-level understanding of how the tools work and the structure and function of the code being differentiated.

Acknowledgments

A portion of this research was performed at the Summer Mathematical Center for Advanced Research in Scientific Computing (CEMRACS) sponsored by the French National Center for Scientific Research, the French Ministry in charge of Universities and Research, and the European Mathematics Society. We thank the organizers, particularly Yvon Maday, for providing a forum for collaborative, interdisciplinary research; the Centre International de Rencontre en Mathématiques for hosting the center; and Radu Lupsa for conducting numerical experiments to identify appropriate parameters for the implicit methods. Thomas Slawig developed the algorithm for wing rotation and conducted the preliminary experiments discussed in Section 6. This work was also supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

References

[1] G. Albada and B. V. Leer. Flux vector splitting and runge-kutta methods for the euler equations. Technical Report 84-27, ICASE, 1984.

[2] B. Averick, J. Moré, C. Bischof, A. Carle, and A. Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[3] Y. Y. Azmy. Post-convergence automatic differentiation of iterative schemes. *Nuclear Science and Engineering*, 125:12–18, 1997.

[4] S. Balay, W. Gropp, L. C. McInnes, and B. Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 - Revision 2.0.17, Argonne National Laboratory, Oct. 1996.

[5] C. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[6] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR 2.0 user's guide (Revision C). Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1995.

[7] C. Faure. Splitting of algebraic expressions for automatic differentiation. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 117–128, Philadelphia, PA, 1996. SIAM.

[8] A. Griewank. On automatic differentiation. In *Mathematical Programming: Recent Developments and Applications*, pages 83–108, Amsterdam, 1989. Kluwer Academic Publishers.

[9] A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992.

[10] A. Griewank, C. Bischof, G. Corliss, A. Carle, and K. Williamson. Derivative convergence of iterative equation solvers. *Optimization Methods and Software*, 2:321–355, 1993.

[11] P. Hovland, C. Bischof, D. Spiegelman, and M. Casella. Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. *SIAM Journal on Scientific Computing*, 18(4):1056–1066, July 1997.

[12] D. Juedes. A taxonomy of automatic differentiation tools. In A. Griewank and G. Corliss, editors, *Proceedings of the Workshop on Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–330, Philadelphia, PA, 1991. SIAM.

[13] V. Korivi, L. Sherman, A. Taylor, G. Hou, L. Green, and P. Newman. First- and second-order aerodynamic sensitivity derivatives via

automatic differentiation with incremental iterative methods. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization,* AIAA 94-4262, pages 87–120. American Institute of Aeronautics and Astronautics, 1994.

[14] B. Mohammadi. CFD with NSC2KE: an user guide. Technical Report 164, INRIA, 1994.

[15] B. Mohammadi. A new optimal shape design procedure for inviscid and viscous turbulent flows. *International Journal for Numerical Methods in Fluids*, 25:183–203, 1997.

[16] S. Osher and F. Solomon. Upwind difference schemes for the hyperbolic systems of conservation laws. *Mathematics of Computation*, 38(158):339–374, 1982.

[17] P.L.Roe. Approximate Riemann solvers, parameters vectors and difference schemes,. *Journal of Computational Physics*, 43, 1981.

[18] N. Rostaing, S. Dalmas, and A. Galligo. Automatic differentiation in Odyssee. *Tellus*, 45a(5):558–568, October 1993.

[19] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing*, 7:856–869, 1986.

[20] R. Struijs, H. Deconinck, P. de Palma, P. Roe, and G.G.Powel. Progress on multidimensional upwind euler solvers for unstructured grids. AIAA paper 91-1550, 1991.