Improving Derivative Performance for CFD Using Simplified Recurrences

Alan Carle
Mike Fagan

CRPC-TR96643
March 1996

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

---

revised November 1996

Improving Derivative Performance for CFD Using Simplified Recurrences*

CRPC-TR96643

Alan Carle Mike Fagan

March 21, 1996

Abstract

Many scientific programs generate solutions to partial differential equations by using an iterative process to reduce a residual quantity. Straightforward differentiation of these programs yields an iterative recurrence that involves both the derivative of the residual, and the derivative of the original process. For a certain class of these problems, the derivative recurrence can be *simplified* to avoid computing the derivative of the original process. A program implementing a simplified recurrence should run faster than its fully differentiated counterpart, since it avoids the redundant computation. In this paper we demonstrate a method for using automatic differentiation tools to facilitate the production of simplified derivative recurrence programs. We illustrate our technique by constructing and validating a simplified derivative version of the OVERFLOW computational fluid dynamics program using the ADIFOR automatic differentiation tool.

1 Introduction

Many engineers are using sensitivity (derivative) calculations to enhance their designs. However, constructing analytic derivatives by hand to compute the needed sensitivities for complex computer models is quite difficult, if not impossible. Consequently, engineers are turning to automatic differentiation (AD) to provide analytic derivatives for their computer models [?]. For a certain class of problems, engineers and numerical analysts have developed a special technique for speeding up the derivative calculation. This technique is referred to by the phrases *simplified recurrences* [?] and *incremental iterative method* [?]. For the sake of consistency, we use "SR" to designate this method. Briefly, the SR technique depends on using analytic derivatives for only *part* of the calculation, as opposed to the entire calculation. Currently, AD tools cannot automatically select the appropriate parts to differentiate, so human interaction is required to produce an SR implementation. A judicious use of AD tools by a human, however, facilitates the construction of SR codes.

Our study of the SR technique followed two lines of inquiry. First, we studied the difficulty of transforming a production quality computer code into the SR form. Since we are not fluid dynamics experts, our experience in constructing a viable SR code should be a fair test of the difficulty in applying the technique. Second, we studied the effectiveness of the SR method itself. We investigated both the accuracy and the efficiency of the SR derivatives by comparing them to conventional AD derivatives.

Our platform for this study was a computational fluid dynamics (CFD) program called OVERFLOW [?]. Our automatic differentiation tool was ADIFOR [?]. The remainder of this paper details the results of our use of ADIFOR to produce an SR implementation of OVERFLOW, and our subsequent study of the

*This work was supported by the National Aerospace Agency under Cooperative Agreement No. NCC 1 212, and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008. Additional computing resources provided by NASA Ames High Performance Computing and Communication Program Testbed 1 Cooperative Research Agreement (through IBM P.O. # 835600Z-E1).

efficacy of the derivatives produced. The paper is divided into four sections. First, we overview the SR technique. Second, we describe the process of converting an arbitrary code into SR form. Third, we discuss our experiences applying this technique to OVERFLOW and examine numerical accuracy and efficiency results for the derivative code. Finally, we describe possible improvements to SR OVERFLOW and some remaining open problems.

2 An Overview of the Simplified Recurrence (SR) Technique

The SR technique applies to computing derivatives for programs that solve equations by iterating a recurrence until some associated residual quantity is "small enough." Many scientific programs, and OVERFLOW in particular, fall into this category. In the following discussion, we give an overview of the SR technique. A more complete (and rigorous) discussion can be found in [?] or [?].

A prototypical recurrence iteration, expressed mathematically is

$$L_n \Delta f_n = R_n$$

where f_n is a current solution estimate, R_n is a residual difference between current and desired aspects of the solution, and L_n is the current linearization of the problem operator. The n-th iteration computes both the residual R_n and the linear operator L_n, and then solves the linear system for Δf_n. This Δf_n is added to the current solution f_n to yield the new solution candidate. This process normally terminates when Δf_n is "small enough."

Differentiating the iterative process yields another iterative process, referred to as the *fully differentiated recurrence*, whose iteration is

$$L_n \Delta f'_n + L'_n \Delta f_n = R'_n.$$

Now suppose that we have iterated a number of times, and f has essentially converged. Under this condition, Δf will be "small." Furthermore, L_n will change "very little" once f has converged. If Δf is "very small" relative to L'_n, then the second term in the sum is close to 0, and may be dropped from the calculation. Likewise, L_n is nearly constant, so the recurrence *simplifies* the derivative *recurrence* to

$$L_* \Delta f'_n = R'_n$$

where L_* indicates value of L when f has converged. This recurrence is referred to as the *simplified recurrence*. Note the similarity of this simplified recurrence to the original undifferentiated recurrence.

This method for computing derivatives has several advantages over the fully differentiated recurrence. First, since there is no need to compute $L'_n \Delta f_n$, the simplified recurrence calculation should be more efficient. Second, since the simplified recurrence uses converged values, it may also have enhanced stability properties that we can exploit. Third, the code in the original program that is used to solve $L\Delta f$ can be used to solve $L_* \Delta f'_n$ by using standard multiple right hand side techniques.

Moreover, since the original solver may have been hand tuned, reusing the original solver for derivative calculations may have a significant impact on the performance of the derivative code. For example, OVERFLOW has been optimized to run well on vector processors.

3 Using AD to Produce the SR form

Ideally, we construct an SR code in two steps. First, we (automatically) differentiate the given function

evaluation code:

Original

initialize inputs I
initialize f_0
do until done
  R_n = residual(I, f_n)
  L_n = operator(I, f_n)
  Δf_n = solver(R_n, L_n)
  $f_{n+1} = f_n + \Delta f_n$
  compute outputs $D(I, f_{n+1})$
end

AD of Original

initialize inputs I,I'
initialize f_0,f_0'
do until done
  R_n, R_n' = g_residual(I, I', f_n, f_n')
  L_n, L_n' = g_operator(I, I', f_n, f_n')
  Δf_n,$\Delta f_n'$ = g_solver(R_n, R_n', L_n, L_n')
  $f_{n+1} = f_n + \Delta f_n$
  $f_{n+1}' = f_n' + \Delta f_n'$
  compute outputs
    $D(I, f_{n+1}), D'(I, I', f_{n+1}, f_{n+1}')$
end

We then construct the SR version from the original and AD-generated codes by first computing the converged values f_*, and L_*. Next we replace the call to g_solver with an explicit loop over each of the independent variables. The loop invokes the original solver on each component of R_n' to compute the derivative update component. The code template for this construction is:

AD of Original

initialize inputs I,I'
initialize f_0,f_0'
do until done
  R_n, R_n' = g_residual(I, I', f_n, f_n')
  L_n, L_n' = g_operator(I, I', f_n, f_n')
  Δf_n,$\Delta f_n'$ = g_solver(R_n, R_n', L_n, L_n')
  $f_{n+1} = f_n + \Delta f_n$
  $f_{n+1}' = f_n' + \Delta f_n'$
  compute outputs
    $D(I, f_{n+1}), D'(I, I', f_{n+1}, f_{n+1}')$
end

SR

run original, retain f_*, L_*
initialize inputs I,I'
initialize f_0'
do until done
  R_n, R_n' = g_residual(I, I', f_*, f_n')
  for each v **in** I
    $\Delta f_n'(v)$ **= solver**($R_n'(v), L_*$)
  end
  $f_{n+1}' = f_n' + \Delta f_n'$
  compute outputs
    $D(I, f_{n+1}), D'(I, I', f_{n+1}, f_{n+1}')$
end

In reality, the degree of difficulty in carrying out the previous steps depends on how the code is structured. A favorably-structured code has two important features. First, the residual calculation, linear operator code, and solver code are all easily identifiable. Second, these three calculations are easily separated from the surrounding code. In software engineering terms, the residual calculation, the linear operator, and the solver must not be *tightly coupled*.

Loosely coupled code often uses many subroutine calls, and few references to global storage. For these codes, conversion to SR form is relatively straightforward. At the other extreme, some programs interleave the calculation of residual, operator and solver. For such an interleaved code, separating the appropriate pieces is a monumental task, and consequently, converting such a code to SR form is equally monumental. Finally, some codes are intermediate in their coupling strength. In these codes, some of the operations are tightly coupled, but others are separable. The codes in this intermediate class exhibit a high degree of coupling at the solver/operator interface, but the residual calculation is well separated from the solver/operator pair. Codes exhibiting this coupling pattern can be easily converted to SR form. We follow similar steps as in the

ideal case, but we retain the coupled operator/solver pair:

Intermediate Coupled Original
initialize inputs I
initialize f_0
do until done
  R_n = residual(I, f_n)
  Δf_n = oper/solv(I, f_n, R_n)
  $f_{n+1} = f_n + \Delta f_n$
  compute outputs $D(I, f_{n+1})$
end

AD of Int. Coupled Original
initialize inputs I,I'
initialize f_0,f'_0
do until done
  R_n, R'_n = g_residual(I, I', f_n, f'_n)
  Δf_n,$\Delta f'_n$ =
    g_oper/solv(I, I', f_n, f'_n, R_n, R'_n)
  $f_{n+1} = f_n + \Delta f_n$
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

From these pieces we construct the SR version from the AD version by:

AD of Int. Original
initialize inputs I,I'
initialize f_0,f'_0
do until done
  R_n, R'_n = g_residual(I, I', f_n, f'_n)
  Δf_n,$\Delta f'_n$ =
    g_oper/solv(I, I', f_n, f'_n, R_n, R'_n)
  $f_{n+1} = f_n + \Delta f_n$
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

SR of Int. Coupled Original
run original, retain f_*
initialize inputs I,I'
initialize f'_0
do until done
  R_n, R'_n = g_residual(I, I', f_*, f'_n)
  for each v **in** I
    $\Delta f'_n(v)$ **= oper/solv**(I', f_*, $R'_n(v)$)
  end
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

The AD derived code can also be used to produce an independent verification program for the SR code. Since the AD generated code computes the function value as well as the derivative, g_residual calculates both R_n and R'_n. Thus, if we set R_n to 0, then the solver for L_n will compute 0 as the value of Δf_n. Since f_n does not change, L_n remains constant. Thus, f_n and L_n remain constant, as they would in the SR iteration. Furthermore, since Δf_n is 0, the general AD calculation of $L_n \Delta f'_n + L'_n \Delta f_n = R'_n$ reduces to $L_n \Delta f'_n = R'_n$, again mimicing the behavior of a "true" SR computation. Consequently, if we start the verifier with the same converged f_*, then it should produce the same iteration sequence as the SR code, albeit a bit more slowly. The code template for the (intermediate coupled) verifier is:

Verify SR
run original, retain f_*
initialize inputs I,I'
initialize f_0,f'_0
do until done
  R_n, R'_n = g_residual(I, I', f_n, f'_n)
  R_n **= 0**
  Δf_n,$\Delta f'_n$ =
    g_oper/solv(I, I', f_n, f'_n, R_n, R'_n)
  $f_{n+1} = f_n + \Delta f_n$
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

4 Our Experiences

To construct an SR version of OVERFLOW, referred to as SR-OVERFLOW, we noted that the OVERFLOW code structure exhibited the intermediate coupling described in the previous section. Then following the steps in the previous section, we applied ADIFOR to the complete OVERFLOW source code to create AD-OVERFLOW. From OVERFLOW and AD-OVERFLOW components, we created both SR-OVERFLOW and the associated SR verification program SRV-OVERFLOW. All programs were compiled with identical optimization flags.

For our tests with OVERFLOW, we differentiated clp and cdp (coefficients of lift and drag due to pressure), clf and cdf (coefficients of lift and drag due to friction), and cmp (pitching moment coefficient) with respect to each of alpha (angle of attack), fsmach (free stream mach number), and rey (reynolds number). In the tables throughout this section we present selected representative derivative values.

The test problem grid we used was the NACA-0012 wing body, with two different aerodynamic configurations. One configuration generated a shock wave, the other did not. We refer to these as the "Shock" and "Noshock" cases, respectively. The shock case is known to be more numerically challenging.

Our study of SR-OVERFLOW had four main components. First, we verified the derivatives generated by AD-OVERFLOW using finite differences. Second, we verified our SR implementation of OVERFLOW by comparing the derivatives computed by SR-OVERFLOW and SRV-OVERFLOW. Third, we verified the accuracy of the SR-OVERFLOW derivatives by comparing them to AD-OVERFLOW derivatives. Finally, we compared the efficiency of SR-OVERFLOW to AD-OVERFLOW.

Our primary computing platform for the accuracy and efficiency study was one of the eight IBM RS6000 nodes comprising the IBM SP2 at Rice[1]. As a secondary computing platform, we had access to a Cray-YMP which we used to gather some numerical evidence about the importance of small residuals for SR calculations.

Finite Differences vs. AD-OVERFLOW. We convinced ourselves that the AD derivatives were valid by doing finite difference calculations, with a suite of different step sizes. The finite difference calculations with an empirically determined step size showed good agreement with the AD derivatives for both the noshock and shock cases. A small sample of our validation results for the shock case appears below. Note that different step sizes are required for different independent variables. The validation of the AD derivatives enabled us to use these values as a check against the SR derivatives. The finite difference and AD-OVERFLOW runs were each 2000 steps in length.

Deriv	AD	FD	Step Size
d(clp)/d(alpha)	-0.84079E-05	-0.8674E-05	.0001
d(cdp)/d(fsmach)	-0.26594E-02	-0.2641E-02	.001
d(clf)/d(rey)	-0.39943E-12	-0.4146E-12	.01

SR-OVERFLOW vs. SRV-OVERFLOW. The outputs of SR-OVERFLOW and SRV-OVERFLOW verified that we had implemented the SR technique correctly (or else that we incorrectly implemented both codes). After iterating the function 2000 steps to obtain f_*, we ran the SR-OVERFLOW and SRV-OVERFLOW codes 100 steps. Both programs produced identical values over this range.

Accuracy of AD-OVERFLOW vs. SR-OVERFLOW. Computing accurate derivatives using SR-OVERFLOW required a good initial derivative iterate f_0'. We generated f_0' by invoked the ADIFOR-generated version of the routine in OVERFLOW that was responsible for constructing the initial iterate f_0. We discovered however that the "slow start" feature of OVERFLOW implicitly coupled the initial value for f_0' and our saved value for f_*, since calling the procedure to compute f_0' reset the iteration counter, causing OVERFLOW to modify f_*. Since the slow start feature deactivates after about 50 steps, we were able to

[1]OVERFLOW version 1.6b is a sequential code. We are currently considering applying the SR technique to a newer parallel version of OVERFLOW.

use 50 steps of AD-OVERFLOW to generate f'_0. The modified SR code template is:

Original SR-OVERFLOW

run original, retain f_*
initialize inputs I,I'
initialize f'_0
do until done
  R_n, R'_n = g_residual(I, I', f_*, f'_n)
  for each v in I
    $\Delta f'_n(v)$ = oper/solv(I', f_*, $R'_n(v)$)
  end
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

Modified SR-OVERFLOW

run original, retain f_*
run 50 steps of AD-OVERFLOW
  retain f'_{50} as f'_0
initialize inputs I,I'
do until done
  R_n, R'_n = g_residual(I, I', f_*, f'_n)
  for each v in I
    $\Delta f'_n(v)$ = oper/solv(I', f_*, $R'_n(v)$)
  end
  $f'_{n+1} = f'_n + \Delta f'_n$
  compute outputs
    $D(I, f_{n+1})$, $D'(I, I', f_{n+1}, f'_{n+1})$
end

On the RS6000, we ran SR-OVERFLOW for 2000 steps, using 2000 steps of the function evaluation and 50 steps of AD (to avoid slow start) for the initial conditions. We ran 2000 steps of AD-OVERFLOW for comparison. Using the modified SR-OVERFLOW, the SR derivatives agreed well with their AD counterparts on the noshock case:

AD noshock (2000 steps)				**SR noshock (2000 steps)**		
var	d/d(alpha)	d/d(fsmach)	d/(rey)	d/d(alpha)	d/d(fsmach)	d/(rey)
clp	0.66148E-01	0.12960	0.26976E-09	0.66159E-01	0.12958	0.26919E-09
cdp	0.38317E-03	0.70816E-03	-0.19670E-10	0.38501E-03	0.70375E-03	-0.19761E-10
clf	-0.10167E-04	0.49246E-04	-0.17396E-11	-0.10171E-04	0.49243E-04	-0.17394E-11
cdf	-0.37907E-04	-0.19311E-03	-0.12562E-09	-0.37744E-04	-0.19335E-03	-0.12563E-09
cmp	-0.15652E-01	-0.22001E-01	-0.10179E-09	-0.15656E-01	-0.21992E-01	-0.10158E-09

For the shock case, the SR and AD derivatives did *not* agree as well on the RS6000.

AD shock (2000 steps)				**SR shock (2000 steps)**		
var	d/d(alpha)	d/d(fsmach)	d/(rey)	d/d(alpha)	d/d(fsmach)	d/(rey)
clp	0.33400E-01	-1.71803	0.89707E-09	0.24091E-01	-1.65581	0.13985E-08
cdp	0.55582E-02	0.12244	0.25632E-10	0.51946E-02	0.12641	0.46293E-10
clf	-0.84079E-05	-0.93477E-04	-0.39943E-12	-0.57789E-05	-0.91523E-04	-0.53150E-12
cdf	-0.63697E-04	-0.26594E-02	-0.99363E-10	-0.90592E-04	-0.26549E-02	-0.98033E-10
cmp	-0.85361E-02	0.36627	-0.32681E-09	-0.37967E-02	0.33726	-0.58010E-09

A look at norm of the residual revealed that the iterations of the shock case had not reduced the residual as much as they had in the noshock case:

Iteration	Shock l_2-Norm	Noshock l_2-Norm
1	.11250E-07	.17661E-07
2000	.38984E-10	.62117E-10

To study this effect further, we ran the original OVERFLOW function iteration on the Cray. The Cray's higher precision computation reduced the residual norm to 0.87797E-11 at step 2000. The SR computation on the Cray agreed well with the AD computation. Since additional steps on the Cray continued to reduce the residual, we ran SR-OVERFLOW for 2000 steps using 3000 steps of the function evaluation and 50 steps of AD.

AD shock (on Cray)				SR shock (on Cray)		
var	d/d(alpha)	d/d(fsmach)	d/(rey)	d/d(alpha)	d/d(fsmach)	d/(rey)
clp	2.59160E-2	-1.66587	1.28353E-9	2.62149E-2	-1.65306	1.26960E-9
cdp	5.23432E-3	0.12523	4.28815E-11	5.25208E-3	0.12620	4.20646E-11
clf	-7.02930E-6	-9.64864E-5	-4.68344E-13	-7.04741E-6	-9.56091E-5	-4.67381E-13
cdf	-7.76275E-5	-2.61447E-3	-9.86958E-11	-7.74364E-5	-2.62202E-3	-9.87064E-11
cmp	-4.56306E-3	0.33916	-5.30172E-10	-4.71294E-3	0.33319	-5.23121E-10

A precondition for applying the SR technique to an iterative code is that the code be able to deliver small enough residuals. Unfortunately, we do not yet know what constitutes "small enough."

Efficiency of AD-OVERFLOW vs. SR-OVERFLOW. The second part of our research studied the efficiency of an SR calculation. To time the relevant portions of the codes under consideration, we used the provided library timing routines (`wtime` on the SP2). For OVERFLOW, AD-OVERFLOW and SR-OVERFLOW, we measured the total time for an iteration and the time required for the operator/solver. The times below are average execution times for a single time step.

program	**operator/solver time**	**total iteration time**
OVERFLOW	3.7 seconds	10.5 seconds
SR-OVERFLOW	14.7 seconds	48.0 seconds
AD-OVERFLOW	17.3 seconds	51.8 seconds

Since the SR program calls the operator/solver for each of the 3 independent variables, the time spent in the operator/solver for a single SR step should be about 3 times the operator/solver cost of a function evaluation. These results show that our implementation of SR for OVERFLOW produces only modest gains over AD.

In light of these modest gains, we investigated another method of improvement suggested by Art Taylor[?]. Taylor suggested that the use of an already converged L_* operator should have increased stability properties, and thus the size of the time step at each iteration could be increased. Increasing the time step would have the effect of lowering the number of iterations necessary to get a final answer. We tested this idea on the shock case. We increased the time steps by factors of 2,4, and 8. In all cases, the answer converged to the correct AD value. The number of iterations was reduced by factors of 2, 4, and *4*. Apparently, increasing the time step by a factor of 8 does not improve the convergence any more than a factor of 4 improves it. See Figure 1 for the convergence plots.
The following table shows the derivative values computed by SR-OVERFLOW using a 4 times longer time step after 500 steps of the derivative iteration.

SR shock (on Cray) with timestep*4

var	d/d(alpha)	d/d(fsmach)	d/(rey)
clp	2.60129E-2	-1.66037	1.24471E-9
cdp	5.17450E-3	0.12472	4.10671E-11
clf	-7.24317E-6	-1.00378E-4	-4.63093E-13
cdf	-7.76998E-5	-2.61274E-3	-9.87155E-11
cmp	-4.54000E-3	0.33721	-5.12204E-10

We do not know if this technique always works, but it *certainly* bears further investigation.

5 Further Directions, and Open Questions

With regard to open technical questions about the efficacy of the SR method, we have identified several important questions bearing further investigation:

- How do you determine when the residual computed by an iterative solver is "small enough" to use the SR technique?

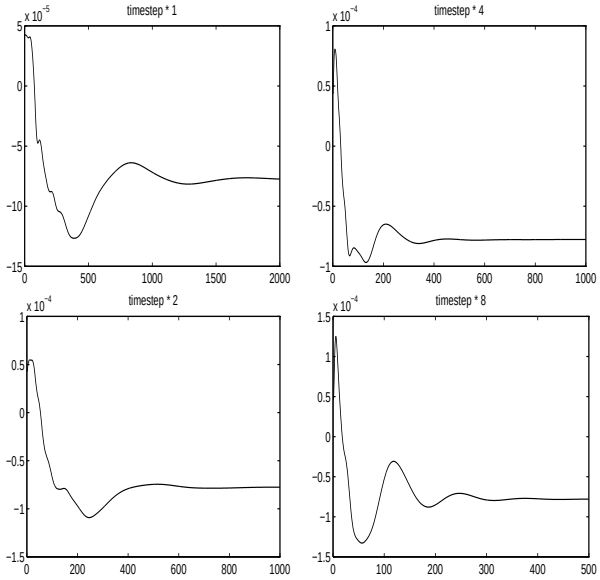

Figure 1 Convergence behavior for SR with increased time steps

- When is it valid to increase the size of the time step for the SR derivative calculation? We think the theory and practice of this technique require greater scrutiny.
- How much improvement in SR derivative code efficiency can we get by putting in additional effort to decouple the operator/solver? Decoupling the solver would allow factoring the linear operator L_n only once. It is unclear how much impact this would have on the total time required to compute derivatives.

6 Summary and Conclusions

In this paper, we reported our study of the use of automatic differentiation as applied to the generation of SR derivative calculations. Our vehicle for study was the CFD code OVERFLOW, and our automatic differentiation tool was ADIFOR. From our study, we reached several conclusions:

- For favorably structured codes, the SR form is not too difficult to construct from an AD generated full recurrence (AD) code.
- An independent verifier for the SR calculations can also be constructed with little additional effort.
- The initial iterate affects the accuracy of an SR calculation. The AD version of the code may be useful in generating favorable initial iterates.
- If the residuals are small, then the SR calculation appears to be as accurate as the AD calculations. Determining when the residual are sufficiently small is an open problem.
- Efficiency improvements were modest. We believe that it is possible to speed up the SR iterations, but the cost in programming time may be prohibitive.
- The greatest promise for efficiency improvement comes from the possibility of increasing the time step for SR calculations. We conjecture that working with converged residuals increases the solver stability, so larger time steps can be taken during each iteration. Our experiments are positive, but this technique merits further theoretical and practical study.

7 Acknowledgments

We would like to offer special thanks to Ray Gomez of NASA Johnson, Perry Newman and Larry Green of NASA Langley, Art Taylor of Old Dominion University, and David Serafini at Rice, for their fluid and aerodynamical insights on this project. We would also like to thank Pieter Buning of NASA Ames for answering all our questions about the OVERFLOW program itself.