Adaptive SOR: A case study in automatic differentiation of algorithm parameters *

Paul Hovland† Michael Heath‡

July 2, 1997

Abstract

Many algorithms make use of one or more parameters to control the behavior of the algorithm. Examples include the damping factor α in a damped Newton method or the relaxation parameter ω in a successive over-relaxation (SOR) iterative solver. The optimal value for such parameters is problem dependent and difficult to determine for most problems. We describe the use of automatic differentiation (AD) to adjust algorithm parameters toward optimal behavior. We demonstrate how AD can be used to transform an SOR solver with fixed ω into an adaptive SOR solver that adjusts ω toward its optimal value. We provide experimental evidence that for large problems with lax convergence criteria such an adaptive solver may converge faster than a solver using an optimal, but fixed, value for ω. These are exactly the conditions that apply when SOR is used as a preconditioner, its most common use in modern scientific computing.

1 Introduction

Many algorithms make use of one or more parameters to control the behavior of the algorithm. Consider, for example, the relaxation parameter ω in a successive over-relaxation (SOR) solver (see [4], for example). An SOR solver uses iterations of the form

$$M_\omega x^{(k+1)} = N_\omega x^{(k)} + \omega b,$$

where

$$M_\omega = D + \omega L$$

and

$$N_\omega = (1-\omega)D - \omega U,$$

where

$$Ax = b$$

is the linear system to be solved and D, L, and U correspond to the diagonal, lower triangular, and upper triangular portions of A, respectively. The parameter ω may be assigned any value in the interval $[0, 2]$, although typically values in the interval $[1, 2]$ are used (hence the term over-relaxation). The value of ω can have a significant effect on the convergence rate of the algorithm. The optimal value for ω can be determined for certain special problems, but in general can be determined only by using an eigenvalue analysis.

*This work was 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.

†Current address: Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439, hovland@mcs.anl.gov.

‡Department of Computer Science, University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, IL 61801, heath@cs.uiuc.edu.

Automatic differentiation (AD) can be used to transform a program that computes a function into a new program that also computes the derivatives of that function. The method works by differentiating individual operations, such as multiplication and square root, for which derivatives can be determined by table lookup, then propagating these derivatives via the chain rule of differential calculus. AD is applied to algorithms, not some abstract notion of a function. Thus, the method used to compute a function can affect the values computed for the derivatives. In certain cases [3, 6], this may affect the quality of the derivatives computed using AD if precautions are not taken. However, it is also possible to exploit this feature by computing derivatives with respect to algorithm parameters.

Computing derivatives with respect to algorithm parameters provides us with information about how changing the values of the parameters would affect the behavior of the algorithm. In the case of an iterative algorithm, we can use this information to adjust the parameters toward values with more desirable behavior. For other algorithms, we can use this information to characterize the effects of the parameters on the behavior of the algorithm, enabling us to develop better heuristics for choosing the parameter values.

2 Automatic Differentiation

Complex functions are often expressed as algorithms and computed using computer programs. Such programs may range from tens to many thousands of lines of code. Automatic differentiation has proven an effective means of developing code to compute the derivatives of such functions. AD relies upon the fact that all programs, no matter how complicated, use a limited set of elementary operations and functions, as defined by the programming language. The function computed by the program is simply the composition of these elementary functions. Thus, we can compute the partial derivatives of the elementary functions using formulas obtained via table lookup, then compute the overall derivatives using the chain rule. This process can be completely automated, and is thus termed *automatic differentiation* [5].

Consider the code for computing the function $y = f(x)$, where $f(x) = (\sin(x)\sqrt{x})/x$, shown in Figure 1(a). Using AD, we can generate code to compute both y and dy/dx, as shown in Figure 1(b). While this example is very simple, AD can be applied to complex programs of arbitrary length. The ADIFOR tool has been applied to programs of over 100,000 lines [1].

```
A = sin(X)
B = sqrt(X)
C = A * B
Y = C/X
```

(a)

```
A = sin(X)
dAdX = cos(X)                    ! table lookup
B = sqrt(X)
dBdX = 1/(2*B)                   ! table lookup
C = A * B
dCdA = B                         ! table lookup
dCdB = A                         ! table lookup
dCdX = dCdA*dAdX + dCdB*dBdX     ! chain rule
Y = C/X
dYdC = 1/X                       ! table lookup
dYdX = dYdC*dCdX - C/(X*X)       ! CR/TL
```

(b)

Figure 1: Code for computing a simple function (a) and code for computing its derivatives generated by AD (b).

An importance consequence of differentiating a function in this manner is that the derivative computation is intertwined with the algorithm used to compute the function. This has implications for the accuracy of the

derivatives, the efficiency of the derivative computation, and the computation of sensitivities. In particular, we can compute the sensitivity of the value computed for the function with respect to algorithm parameters.

3 Automatic Differentiation of Algorithm Parameters

Because AD is applied to a program, we can treat algorithm parameters as independent variables. Doing so enables us to compute the sensitivity of a function with respect to these parameters. These sensitivities are most useful for functions that provide a measure of the quality of the solution. For example, in the case of an SOR iteration, we could compute the derivative of the residual with respect to ω.

The derivative provides information about how using a different value for the algorithm parameter would have affected the quality of the solution. In an iterative algorithm, we can perform the next iteration using values for this parameter that would have provided a better solution for the previous iteration. We thus use information about past behavior of the algorithm to attempt to improve future behavior of the algorithm. Alternatively, we could compute the sensitivities several times for various parameter values in order to characterize the behavior of the algorithm as a function of the algorithm parameters.

4 Adaptive SOR

We now consider a concrete example of using AD to compute derivatives with respect to algorithm parameters and the use of these derivatives to adjust the behavior of the algorithm. The SOR iteration for the finite difference discretization of the Poisson problem

$$\nabla^2 u(x,y) = f(x,y)$$

is

```
for i = 1 to M do
    for j = 1 to N do
        u^GS_{i,j} = (f_{i,j} + u^{k+1}_{i-1,j} + u^k_{i+1,j} + u^{k+1}_{i,j-1} + u^k_{i,j+1})/4
        u^{k+1}_{i,j} = (1 - ω) u^k_{i,j} + ω u^GS_{i,j}.
    enddo
enddo
```

The residual r for this problem is

$$r = \|b - Ax\|_2,$$

where

$$\|b - Ax\|_2^2 = \sum_{i=1}^{M}\sum_{j=1}^{N}(4u_{i,j} - u_{i-1,j} - u_{i+1,j} - u_{i,j-1} - u_{i,j+1} - f_{i,j})^2.$$

We can use AD to produce code for computing the sensitivity of the residual with respect to the relaxation parameter, $\partial r/\partial\omega$, denoted by r'. We can then use the derivative computed for iteration k to adjust the value of ω used for iteration $k+1$. One option is to use Newton's method to try to drive the residual to zero,

$$\omega_{k+1} = \omega_k - \frac{r_k}{r'_k}.$$

However, in general the residual will never equal zero using an iterative method and finite precision arithmetic. An alternative is to use a secant method to minimize the residual (drive r' to zero),

$$\omega_{k+1} = \omega_k + r'_k \frac{\omega_k - \omega_{k-1}}{r'_k - r'_{k-1}}.$$

5 Experimental Results

We implemented the SOR algorithm described above in Fortran, using $f(x, y) = 0$ (the Laplace equation) and Dirichlet boundary conditions, and terminating after the relative residual was below some threshold $\hat{r}$. We applied ADIFOR [1] to this program, then added code that uses the resulting derivatives to modify ω using a secant method. To avoid large changes in ω when r' is small and to keep ω within the required interval, we used the modified form:

$$\begin{aligned} \Delta\omega &= \min\left(.05, r'_k \frac{\omega_k - \omega_{k-1}}{r'_k - r'_{k-1}}\right), \\ \omega_{k+1} &= \max(0, \min(1.985, \omega_k + \Delta\omega)). \end{aligned}$$

The experimentally determined value 1.985 was used to avoid getting "stuck" at 2.

The optimal fixed ω for the Poisson problem can be computed using an analytic formula given in [2] (see [7] for background theory):

$$\omega_{\text{opt}} = \frac{2}{1 + \sqrt{1 - \rho^2}},$$

where

$$\rho = \frac{1}{2}\left(\cos\frac{\pi}{M+1} + \cos\frac{\pi}{N+1}\right).$$

We compared the standard SOR program using this optimal ω to the adaptive algorithm for various problem sizes. Both programs were compiled using `xlf -O` and executed on an IBM SP1 node. Figure 2 shows the progression of the residual and, for the adaptive method, ω over successive iterations for a 300×300 problem. Table 1 shows our results for various problem sizes M, N and stopping thresholds $\hat{r}$ ranging from 10^{-2} to 10^{-4}. We report the residual r_0 for the initial guess $u = 0$ as well as the number of iterations required (its) and the execution time (t) in seconds for fixed $\omega = 1.5$, fixed $\omega = \omega_{\text{opt}}$, and variable ω with starting value 1.5. The program using fixed ω attempts to reduce the overhead of computing r/r_0 by computing it only once every 10 iterations.

From these results, we can conclude that adaptive SOR based on derivatives computed using AD is preferable to regular SOR using fixed ω, even when the optimal value for ω is used. This is especially true for large problems with lax convergence criteria. These are exactly the conditions that apply when SOR is used as a preconditioner, its most common use in modern scientific computing. Large problem sizes help amortize the overhead associated with the computation used to adjust ω. As is illustrated in Figure 2, the optimal value for ω is optimal only asymptotically, so the adaptive method does considerably better when we use lax convergence criteria.

6 Summary

Many algorithms used in scientific computing rely upon parameters that control the behavior of the algorithm. An example is the relaxation parameter ω of the SOR iteration. It is often difficult to determine the optimal value of the parameter for a particular problem. We demonstrated how sensitivities computed using automatic differentiation can be used to adjust an algorithm parameter toward its optimal value. We showed how this technique can be applied to SOR to develop an adaptive SOR algorithm that, even when started at a highly suboptimal ω, outperforms standard SOR using the optimal value for ω.

References

[1] 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.

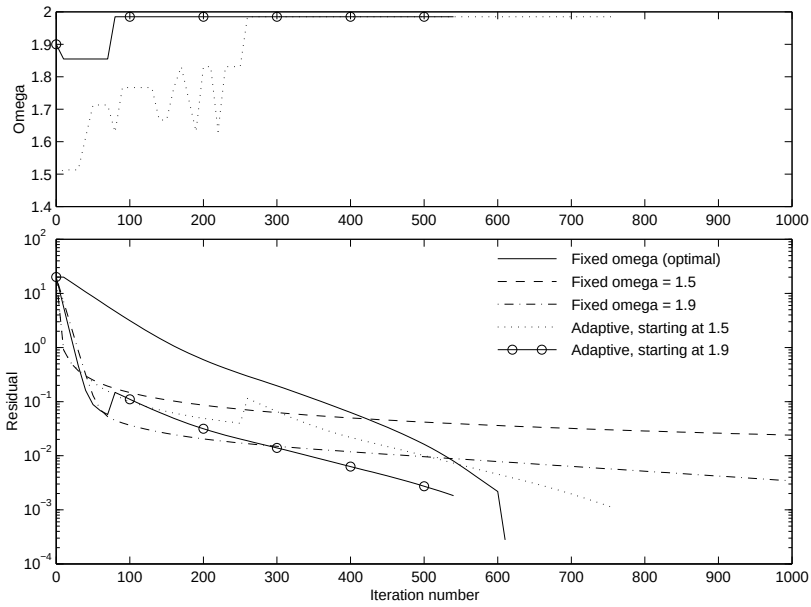

Figure 2: Comparison of methods using starting $\omega > 1$

					Fixed $\omega = 1.5$		Fixed $\omega = \omega_{\text{opt}}$		Adaptive	
M	N	ω_{opt}	r_0	$\hat{r}$	its	t	its	t	its	t
300	300	1.975	20.08	10^{-2}	70	2.15	300	8.59	56	3.65
400	400	1.984	23.17	10^{-2}	70	4.28	400	20.13	58	7.45
500	500	1.987	25.88	10^{-2}	70	7.73	500	39.29	57	12.76
750	750	1.992	31.68	10^{-2}	70	22.41	750	131.61	57	35.06
1000	1000	1.994	36.56	10^{-2}	70	45.58	990	307.45	57	69.68
300	300	1.975	20.08	10^{-3}	1260	36.02	490	13.97	398	23.95
400	400	1.984	23.17	10^{-3}	1300	66.02	650	32.71	330	35.95
500	500	1.987	25.88	10^{-3}	1340	107.39	810	63.55	281	49.29
750	750	1.992	31.68	10^{-3}	1390	258.85	1220	213.75	280	118.45
1000	1000	1.994	36.56	10^{-3}	1410	489.25	1620	502.58	260	208.38
300	300	1.975	20.08	10^{-4}	7910	224.94	610	17.33	676	40.46
400	400	1.984	23.17	10^{-4}	11700	588.78	810	40.82	596	63.91
500	500	1.987	25.88	10^{-4}	15430	1210.96	1010	79.18	608	102.77
750	750	1.992	31.68	10^{-4}	23130	4067.20	1510	264.78	765	295.47
1000	1000	1.994	36.56	10^{-4}	26800	8362.84	2010	623.48	766	535.36

Table 1: Comparison of SOR algorithms using fixed and variable ω.

[2] B. N. Datta. *Numerical Linear Algebra and Applications*. Brooks/Cole, Pacific Grove, CA, 1995.

[3] P. Eberhard and C. Bischof. Automatic differentiation of numerical integration algorithms. Preprint ANL/MCS-621-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[4] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

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

[6] 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.

[7] R. S. Varga. *Matrix Iterative Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1962.