Using ADIFOR 1.0 to Compute Hessians

by

Paul Hovland
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Using ADIFOR 1.0 to Compute Hessians

by

Paul Hovland*

Mathematics and Computer Science Division

Technical Memorandum No. 195

April 1995

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; through NSF Cooperative Agreement No. CCR-8809615; by a U.S. Department of Defense NDSEG Fellowship; and by NASA Purchase Order L25935D.

*Current address: Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2412 Digital Computer Lab, 1304 W. Springfield Ave., Urbana, IL 61801

[Handwritten note: DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED]
## Contents

Abstract 1

1 Introduction 1

2 Procedure 1

3 Discussion 2

4 Initialization 3

5 Example 4

6 Conclusions 8

Appendix: ADIFOR-generated Subroutine for Computing Hessians 9

References 12
Using ADIFOR 1.0 to Compute Hessians

by

Paul Hovland

Abstract

ADIFOR provides a simple means to produce code for the first derivatives of functions through the technique of automatic differentiation. However, the fact that ADIFOR currently cannot produce code to compute second derivatives limits its usefulness for certain applications. This paper describes how ADIFOR and related tools can be used to produce code that does compute second derivatives and discusses how to use this code. Conclusions are presented about the limitations of this method and how it might compare with second-derivative code produced directly by ADIFOR.

1 Introduction

When a scientist wishes to compute the gradient of a function for optimization or sensitivity analysis, ADIFOR provides a simple means to produce derivative code via automatic differentiation [1]. However, many optimization methods require the Hessian of the objective function. Currently, ADIFOR does not produce code for second derivatives. But, by applying ADIFOR twice, it is possible to produce code for the Hessian. The next section outlines a procedure for creating code for second derivatives. Section 3 provides more detailed discussion of why certain steps are required and how they should be executed. Section 4 discusses seed matrix initialization as it applies to this second derivative code. Section 5 provides a simple example of the technique. The final section discusses some of the limitations of the technique and describes how the resultant code might resemble and differ from the code that would be produced by a future version of ADIFOR.

2 Procedure

This paper assumes that the user is already familiar with ADIFOR and the various files involved in its use. Those readers not familiar with ADIFOR are referred to [3,4] for an introduction.

The procedure required to produce code capable of computing second derivatives is as follows:

1. Create an ADIFOR script (.adf) file. Be sure to include a “SEP .” line.
2. Create a composition (.comp) file.
3. Run ADIFOR (adif or func.adf func.comp).
4. Run make on the ADMakefile.
5. Create a main program to call the new top-level subroutine.
6. Create a new composition file, including the new main program, all ADIFOR-generated subroutines, and, if necessary, intrinsic.f.
7. If `intrinsic.f` is not needed (there are no intrinsic functions requiring the exception handler), skip to Step 11.

8. Run `adpre` using this composition file.

9. Edit the resulting `*.ad.f` files so that `$` is changed to `_`.

10. Change the `.ad.f` extension to `.f`, or modify the composition file to use the new file names.

11. Create a new ADIFOR script file, changing the separator (for clarity), and prefacing the OUT variable(s) with `g_`.

12. Run ADIFOR again (`adif` or `g-func.adf`, `gfunc.comp`).

13. Run `make` on the ADMakefile.

14. Create a new main program that does the proper initializations and calls `g$g.func`.

3 Discussion

The reason for some of the steps in Section 2 may not be obvious. This section explains the motivation behind these procedures. It also includes brief notes about the tools being used.

ADIFOR employs a two-stage process to produce derivative code. In the first step, ADIFOR analyzes the program, stores intermediate information, and creates a makefile. This makefile invokes Adtrans, the ADIFOR translator, which translates the intermediate information into source code. The default separator character of `$` causes problems for the make utility. This is normally not a problem, because of the naming convention for the files created by ADIFOR, basically the original subroutine name followed by an extension. However, when ADIFOR is applied twice, the subroutine names from the first application contain the separator character. Thus, the separator character for the first application of ADIFOR must be something other than `$`. For this reason, we suggest a separator character of `_` in Step 1.

The exception-handling routines [2] provide an impediment to applying ADIFOR a second time because these routines are implemented as functions, but ADIFOR currently deals only with subroutines. Steps 8–10 describe how the ADIFOR preprocessor, Adpre, can be used to overcome this problem. However, if the exception-handling routines, found in the file `intrinsic.f`, are not needed these steps can be omitted, as mentioned in Step 7. The ADIFOR preprocessor is described in [4], but its use can be summarized by three steps:

1. Set the `RN_HOME` environment variable using `setenv RN_HOME /anydir`.

2. Create a composition file, as with ADIFOR.

3. Execute `Adpre`, specifying the composition file name, as for example, `adpre -P g.func.comp`.

For similar reasons to those discussed above, the `$` character that occurs in the files produced by Adpre (which have the `.f` extension replaced by `.ad.f`) must replaced by another character, such as `_`.

The Hessian is essentially the derivatives of the gradient with respect to the independent variable. Thus, the ADIFOR script file for the second application of ADIFOR should specify the top-level ADIFOR-generated subroutine as `TOP`, the gradient as `OUT`, and the same `IN` variables(s). In order to make it easier to distinguish between the gradient objects of the first pass and the gradient objects of the second pass, and also to prevent name conflicts, a different separator character (such as `$`) should be used.
4 Initialization

The initialization of seed matrices is nontrivial even for first-derivative programs [3]. In the case of second derivatives, the situation can become even more complex. The code produced by the method outlined above is capable of computing the matrix product $S_I \times H \times S_T^T$, where $H$ is the Hessian. The two seed matrices, $S_I$ and $S_T$, arise from the double application of ADIFOR. If $x$ is the only independent variable, these seed matrices will have the names $g.x$ and $g$x. If all that is desired is the Hessian, these seed matrices should be initialized to an identity matrix, for example by using the code shown in Figure 1.

If only an $m \times p$ region of the Hessian is desired, this can be computed by initializing $g$x and $g.x$ such that the appropriate $p$ elements of $g$x and $m$ elements of $g$ are computed. A special case of this situation arises when a particular column or row of the Hessian is desired. The third column of the Hessian could be computed by using the initialization in Figure 2. Note that this is the transpose of the column because of the way ADIFOR stores derivatives. Since the Hessian is symmetric, this is less important.

The ability to compute one column at a time can be used to exploit the symmetry of the Hessian. An example of how this might be accomplished is shown in Figure 3. However, it is important to note that the overhead of recomputing the function and a portion of the gradient on each iteration implies that this method will not be significantly more efficient than the full Hessian computation and may even be more expensive.

Perhaps the most promising application for seed matrix initialization is when the pre- and post-multiplication of the Hessian by a vector or pair of vectors is desired. For example, if we wish to compute $z^THy$, the simple initialization is sufficient. This capability may be particularly useful for optimization techniques like the conjugate gradient method.

```plaintext
do i = 1, n  
g$x(1,i) = z(i)  
g.x(1,i) = y(i)  
endo  
g$x = 1  
g.x = 1
```

```plaintext
do i = 1, n  
do j = 1, n  
g.x(j, i) = 0.0  
g$x(j, i) = 0.0  
endo  
g.x(i, i) = 1.0  
g$x(i, i) = 1.0  
endo  
g$p$ = n  
g.p = n
```

Figure 1: Seed matrix initialization using identity matrices to compute the Hessian
do i = 1, n
do j = 1, n
g_x(j, i) = 0.0
enddo
g_x(i, i) = 1.0
g$x(1, i) = 0.0
enddo
g$x(1,3) = 1.0
g$p$ = 1
g_p = n

Figure 2: Seed matrix initialization using \( n \times n \) and \( 1 \times n \) matrices to compute a column of the Hessian

do k = 1, n
do i = 1, n
do j = 1, k
g_x(j, i) = 0.0
enddo
g_x(i, i) = 1.0
g$x(1, i) = 0.0
enddo
g$x(1,k) = 1.0
g$p$ = 1
g_p = k
call g$p$_.func(....)

Figure 3: Algorithm using column computations to exploit the Hessian's symmetry

5 Example

Consider the example program in Figure 4. This program simply computes the function

\[
f = \sqrt{\prod_{i=1}^{n} z_i}.
\]

An ADIFOR script file for this program is

IN x
OUT f
TOP func
PMAX 5
SEP _
Figure 4: A simple example program

and an appropriate composition file is

```
func.f
main.f
```

Applying ADIFOR yields a subroutine `g_func.3` (listed in the Appendix) capable of computing the gradient of this function. An appropriate main program for this subroutine is shown in Figure 5.

A composition file for the gradient program is

```
g_main.f
func.3.f
intrinsic.f
```

If Adpre is executed on this composition, several changes occur. The most significant changes involve converting the functions in `intrinsic.f` to subroutines and modifying the calls in `g_func.3` accordingly. For example, the line

```
fbar = g$sqrt(f, r_1)
```

in `g_func.3` becomes

```
call g_sqrtsubr(f, r_1, fbar)
```

after processing with Adpre and the global replacement of `$` by `...`
program example

real x(5), f, g, x(5, 5), g_f(5)
integer i, j

x(1) = 1.0
x(2) = 3.0
x(3) = 2.0
x(4) = 6.0
x(5) = 4.0

do i = 1, 5
   do j = 1, 5
      g_x(j, i) = 0.0
   enddo
   g_x(i, i) = 1.0
enddo

call g_func_3(5, x, g_x, 5, f, g_f, 5, 5)

do i = 1, 5
   write (*, *) 'x(', i, ') =', x(i)
enddo
write (*, *) 'f = ', f

do i = 1, 5
   write (*, *) 'g_f(', i, ') =', g_f(i)
enddo
end

Figure 5: An example program using the ADIFOR-generated gradient subroutine
program example

real x(5), f, g_x(5, 5), g_f(5), g$g_x(5, 5), g$g_f(5, 5), g$g_f(5)
in integer i, j

x(1) = 1.0
x(2) = 3.0
x(3) = 2.0
x(4) = 6.0
x(5) = 4.0

do i = 1, 5
  do j = 1, 5
    g_x(j, i) = 0.0
    g$g_x(j, i) = 0.0
  enddo
  g_x(i, i) = 1.0
  g$g_x(i, i) = 1.0
enddo

call g$g_func_3$d32(5, 5, x, g_x(5), g_x(5), f, g$f(5, 5), g$f(5), g$f(5, 5, 5, 5))

do i = 1, 5
  write (*, *) 'x(', i, ') = ', x(i)
enddo
write (*, *) 'f = ', f
do i = 1, 5
  write (*, *) 'g_f(', i, ') = ', g_f(i)
enddo

do i = 1, 5
  do j = 1, 5
    write (*, *) 'Hess(', i, ', ', j, ') = ', g$g_f(j, i)
  enddo
enddo
end

Figure 6: An example program using the ADIFOR-generated Hessian subroutine
With an ADIFOR script file such as

```
IN x
OUT g_f
PMAX 5
TOP g_func_3
SEP $
```

ADIFOR can be applied a second time to yield the Hessian code. As was discussed in Section 4, the driver code for this subroutine can initialize variables in a number of different ways. The simplest scheme, where the entire Hessian is computed at once, is used in the main program in Figure 6.

6 Conclusions

Even though ADIFOR does not currently support second derivatives, it is possible to produce code to compute a Hessian by using two passes of ADIFOR. This approach is applicable to all sorts of second derivatives, not just Hessians, but the example of a Hessian has been used for simplicity. The approach described suffers from certain limitations. Foremost is the restriction that the symmetry of the Hessian is not exploited, and unneeded computations are performed. Additional overhead comes from computing the gradient twice. When second-derivative capabilities are built into ADIFOR, it is desirable that this overhead be eliminated. At the same time, there are certain characteristics of the code generated by this procedure that would be desirable in an ADIFOR implementation. In particular, the ability to compute a rectangular region of the Hessian or a matrix-Hessian-matrix product via special seed matrix initializations could be of great benefit to a computational scientist.
Appendix: ADIFOR-generated Subroutine for Computing Hessians

DISCLAIMER

This file was generated on 05/24/94 by the version of ADIFOR compiled on 07/13/93.

ADIFOR was prepared as an account of work sponsored by an agency of the United States Government, Rice University, and the University of Chicago. Neither the author(s), the United States Government nor any agency thereof, nor Rice University, nor the University of Chicago, including any of their employees or officers, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information or process disclosed, or represents that its use would not infringe privately owned rights.

subroutine g$f_g_func_3$32(g$p$, g$p_$, x, g$x$, ldg$x$, g$x$, ldg$x$, f,
  * g$f$, ldg$f$, g$f$, g$f_g$, ldg$g_f$, ldg$f$, n)

Formal g$_f$ is active.
Formal f is active.
Formal x is active.

integer g$p$
integer g$p_{max}$
parameter (g$p_{max}$ = 5)
integer g$i$
real fbaar
real r$1$
integer ldg$x$
integer ldg$f$
integer ldg$g_f$

Formal f is active.
Formal x is active.

integer g$_p_$
integer g$_p_{max}_$
parameter (g$_p_{max}_$ = 5)
integer g$_i_$
real r$_1$
real g$r_1$(g$p_{max}$)
real fbaar
real g$fbaar$(g$p_{max}$)
integer ldg$x$
integer ldg$f$

integer n, i
real f, x(n)
real g$f (ldg$f), g$x(ldg$x, n)
real g_f (ldg_f), g_x(ldg_x, n)
real g$g_f (ldg$g_f, ldg_f)
real g_sqrt
external g_sqrt
real g$sqrt
external g$sqrt
if (g$p$ .gt. g$p$max$) then
    print *, 'Parameter g$p is greater than g$p$max.'
    stop
endif
if (g_p_ .gt. g_p_max_) then
    print *, 'Parameter g_p is greater than g_p_max.'
    stop
endif
f = 1.0
do 99979 g$i$ = 1, g$p$
g$f (g$i$) = 0.0
99979 continue
do 99991, g-i-
    g-f (g-i-1) = 0.0
99991 continue
99995 continue
99999 continue
99993 continue
do 99989, i = 1, n
    f = f * x(i)
    r_1 = x(i)
do 99977 g$i$ = 1, g$p$
g$r,l(g$i$, g-i-) = g$x(g$i$, i)
g-f(g-i-) = r_1 * g$r,l(g$i$, g-i-) + f * g-x(g$i$, g-i-, i)
99977 continue
r_1 = x(i)
do 99990, g-i_ = 1, g_p_
    g_f (g_i_) = r_1 * g_f (g_i_) + f * g_x (g_i_, i)
r$1 = g_f (g_i_)
    fbaar = g_x (g_i_, i)
do 99976 g$i$ = 1, g$p$
g$g_f(g$i$, g_i_) = r$1 * g$g_f(g$i$, g_i_) + fbaar * g$sqrt(g$i$)
++ r_1 * g$g_f(g$i$, g_i_)
99976 continue
    g_f (g_i_) = r_1 + r$1 + f * g_x (g_i_, i)
99993 continue
99998 continue
99990 continue
f = f * r_1
do 99975 g$i$ = 1, g$p$
g$f (g$i$) = r_1 * g$f (g$i$) + f * g$r,1(g$i$)
99975 continue
f = f * r_1
99994 continue
99997 continue
99989 continue
\[ f = \sqrt{f} \]
\[ r_1 = \sqrt{f} \]
\[ r_1 = \sqrt{f} \]
\[ \text{do } 99974 \ g_1(i) = 1, \ g_p \]
\[ g_{r-1}(g_1(i)) = fbaar \ast g_1'(g_1(i)) \]
\[ 99974 \ \text{continue} \]
\[ r_1 = r_1 \]
\[ \text{call } g_{g_1\text{sqrtsubr7}}(g_p, f, g'(1), \lg f, r_1, g_{r-1}(1), g_{pmax}) \]
\[ *x, fbaar, g_{fbar}(i), g_{pmax} \]
\[ \text{do } 99988, \ g_{i-1}, = 1, \ g_p \]
\[ g_{f}(g_{i-1}) = fbaar \ast g_{f}(g_{i-1}) \]
\[ g_{r-1} = g_{f}(g_{i-1}) \]
\[ g_{r-1} = 1, \ g_p, g_{g_{f}(g_{i-1}, g_{i-1})} = r_1 \ast g_{fbaar}(g_{i-1}) + fbaar \ast g_{g_{f}(g_{i-1}, g_{i-1})} \]
\[ 99974 \ \text{continue} \]
\[ 99992 \ \text{continue} \]
\[ 99996 \ \text{continue} \]
\[ 99988 \ \text{continue} \]
\[ i = r_1 \]
\[ \text{do } 99972 \ g_{i-1} = 1, \ g_p, g_{f}(g_{i-1}) = g_{r-1}(g_{i-1}) \]
\[ 99972 \ \text{continue} \]

\text{return} \\
\text{end}
References


