# Using ADIFOR 1.0 to Compute Hessians

*Paul Hovland*

**CRPC-TR95540-S**
**February 1995**

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

# 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 which does compute second derivatives and discusses how to use this code. Conclusions about the limitations of this method and how it might compare to second derivative code produced directly by ADIFOR are presented.

## 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:

1. Create an ADIFOR script (.adf) file. Be sure to include a "SEP _" line.
2. Create a composition (.comp) file.
3. Run ADIFOR (`adifor 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`.

[*]This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy 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.

[†]Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Avenue, Argonne, IL 60439.

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 (`adifor g_func.adf g_func.comp`).

13. Run make on the ADMakefile.

14. Create a new main program which 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 attempts to explain the motivation behind these procedures. It also includes some 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 3 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). 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 things (can) become even more complex. The code produced by the method outlined above is capable of computing the matrix product $S_1 \times H \times S_2^T$, where $H$ is the Hessian. The two seed matrices, $S_1$ and $S_2$, 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 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$f** and $m$ elements of **g_f** are computed. A special case of this situation is when a particular column or row of the Hessian is desired. The third column of the Hessian could be computed 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 a 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 $z^T H y$, the simple initialization

```
do i=1, n
  g$x(1,i) = z(i)
  g_x(1,i) = y(i)
enddo
g$i$ = 1
g_i_ = 1
```

is sufficient. This capability may be particularly useful for optimization techniques like the conjugate gradient method.

```
do i = 1, n
  do j = 1, n
    g_x(j, i) = 0.0
    g$x(j, i) = 0.0
  enddo
  g_x(i, i) = 1.0
  g$x(i, i) = 1.0
enddo
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$g_func(....)
enddo
```

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

$$f = \mathrm{sqrt}(\prod_{i=1}^{n} x_i).$$

An ADIFOR script file for this program is

```
IN x
OUT f
TOP func
PMAX 5
SEP _
```

```
program example                            subroutine func(x,f,n)

real x(5),f                                integer n,i
integer i                                  real f,x(n)

x(1) = 1.0                                 f=1.0
x(2) = 3.0                                 do i=1,n
x(3) = 2.0                                    f=f*x(i)
x(4) = 6.0                                 enddo
x(5) = 4.0
                                           f = sqrt(f)
call func(x,f,5)
                                           return
do i=1,5                                   end
  write(*,*) 'x(',i,') = ',x(i)
enddo
write(*,*) 'f = ',f

end
```

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 _.

Using an ADIFOR script file such as

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

```
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$x(5,5),g$g_f(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
    g$x(j, i) = 0.0
  enddo
  g_x(i, i) = 1.0
  g$x(i, i) = 1.0
enddo

call g$g_func_3$32(5,5,x,g$x,5,g_x,5,f,g$f,5,g_f,g$g_f,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

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.

## References

[1] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[2] Christian Bischof, George Corliss, and Andreas Griewank. ADIFOR exception handling. Technical Report ANL/MCS-TM-159, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[3] Christian Bischof and Paul Hovland. Using ADIFOR to compute dense and sparse Jacobians. Technical Report ANL/MCS-TM-158, Mathematics and Computer Science Division, Argonne National Laboratory, 1991.

[4] Christian H. Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. Getting started with ADIFOR. Technical Report ANL/MCS-TM-164, Mathematics and Computer Science Division, Argonne National Laboratory, 1992.

## Appendix: ADIFOR-generated subroutine for computing Hessian

```
C                                DISCLAIMER
C
C     This file was generated on 05/24/94 by the version of
C     ADIFOR compiled on 07/13/93.
C
C     ADIFOR was prepared as an account of work sponsored by an
C     agency of the United States Government, Rice University, and
C     the University of Chicago.  Neither the author(s), the United
C     States Government nor any agency thereof, nor Rice University,
C     nor the University of Chicago, including any of their employees
C     or officers, makes any warranty, express or implied, or assumes
C     any legal liability or responsibility for the accuracy, complete-
C     ness, or usefulness of any information or process disclosed, or
C     represents that its use would not infringe privately owned rights.
C
      subroutine g$g_func_3$32(g$p$, g_p_, x, g$x, ldg$x, g_x, ldg_x, f,
     * g$f, ldg$f, g_f, g$g_f, ldg$g_f, ldg_f, n)
C
C       Formal g_f is active.
C       Formal f is active.
C       Formal x is active.
C
        integer g$p$
        integer g$pmax$
        parameter (g$pmax$ = 5)
        integer g$i$
        real fbaar
        real r$1
        integer ldg$x
        integer ldg$f
        integer ldg$g_f
C
C
C       Formal f is active.
C       Formal x is active.
C
        integer g_p_
        integer g_pmax_
        parameter (g_pmax_ = 5)
        integer g_i_
        real r_1
        real g$r_1(g$pmax$)
        real fbar
        real g$fbar(g$pmax$)
        integer ldg_x
        integer ldg_f
C

        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$pmax$) then
        print *, 'Parameter g$p is greater than g$pmax.'
        stop
      endif
      if (g_p_ .gt. g_pmax_) then
        print *, 'Parameter g_p is greater than g_pmax.'
        stop
      endif
      f = 1.0
      do 99979 g$i$ = 1, g$p$
        g$f(g$i$) = 0.0
99979 continue
      do 99991, g_i_ = 1, g_p_
        g_f(g_i_) = 0.0
        do 99978 g$i$ = 1, g$p$
          g$g_f(g$i$, g_i_) = 0.0
99978   continue
99995   continue
99999   continue
99991 continue
      do 99989, i = 1, n
C       f = f * x(i)
C       r_1 = x(i)
        do 99977 g$i$ = 1, g$p$
          g$r_1(g$i$) = g$x(g$i$, i)
99977   continue
        r_1 = x(i)
        do 99990, g_i_ = 1, g_p_
C         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$1 * g$r_1(g$i$) + fbaar * g$f(g$i$)
     *+ r_1 * g$g_f(g$i$, g_i_)
99976     continue
          g_f(g_i_) = r_1 * r$1 + f * g_x(g_i_, i)
99993     continue
99998     continue
99990   continue
C       f = f * r_1
        do 99975 g$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
```

```
C       f = sqrt(f)
C       r_1 = sqrt(f)
        r$1 = sqrt(f)
        fbaar = g$sqrt(f, r$1)
        do 99974 g$i$ = 1, g$p$
          g$r_1(g$i$) = fbaar * g$f(g$i$)
99974   continue
        r_1 = r$1
        call g$g_sqrtsubr$7(g$p$, f, g$f(1), ldg$f, r_1, g$r_1(1), g$pma
     *x$, fbar, g$fbar(1), g$pmax$)
        do 99988, g_i_ = 1, g_p_
C         g_f(g_i_) = fbar * g_f(g_i_)
          r$1 = g_f(g_i_)
          do 99973 g$i$ = 1, g$p$
            g$g_f(g$i$, g_i_) = r$1 * g$fbar(g$i$) + fbar * g$g_f(g$i$,
     *g_i_)
99973     continue
          g_f(g_i_) = fbar * r$1
99992     continue
99996     continue
99988   continue
        f = r_1
        do 99972 g$i$ = 1, g$p$
          g$f(g$i$) = g$r_1(g$i$)
99972   continue

        return
      end
```