A Comparison of ADIFOR-Generated versus Hand-Generated Derivatives for a Complicated Statistical Function

Alan Carle
Mike Fagan

CRPC-TR95526
April 1995

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

This work was supported in part by the National Aerospace Agency under Cooperative Agreement No. NCCW-0027 and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

A Comparison of ADIFOR-Generated versus Hand-Generated Derivatives for a Complicated Statistical Function[1]

Alan Carle
carle@cs.rice.edu

Mike Fagan
mfagan@cs.rice.edu

Rice University MS 41
Center for Research on Parallel Computation
6100 S. Main Street
Houston, TX 77251-1892

CRPC Technical Report CRPC-TR95526

Key words. Automatic differentiation, ADIFOR, derivative, ADIntrinsics, LLDRLF.

Abstract. This report compares results computed by automatic differentiation (via ADIFOR) and by hand-coded derivatives for a numerically complicated statistical code. This report analyzes the types of discrepancies that were found and describes steps taken to mediate each of them.

1 Introduction

This report describes the application of the ADIFOR 2.0 automatic differentiation system[1, 2] to produce code to compute derivatives for a complicated statistical function: the log-likelihood for log-F distribution (LLDRLF)[3]. The ADIFOR system generated code to compute first and second derivative functions from the statistical function code. The authors of the statistical function code also "hand-coded" a version of the derivatives.

The likelihood function calculation is exceptionally difficult to calculate numerically due to an exceptionally wide input domain[3, 4]. Consequently, the strategy for computing this function is complicated. One of five different methods (Extreme Value Method, Bratio Method, Asymptotic Method, Continued Fraction Method, Series Method) is used to compute the function approximation within regions of the input domain. Thus, numerical accuracy is maintained over the entire space of input values. The hand-coded derivatives follow the same basic strategy – one of five different methods (Extreme Value Method, Asymptotic Method, Continued Fraction Method, Series Method, Direct Differentiation Method) is used to compute the derivative approximation. Although four of the function approximation methods (all but the Bratio Method) correspond directly to four of the derivative approximation methods (all but the Direct Differentiation Method), the authors of LLDRLF partitioned the input domain for the function and derivative computations differently. Table 1 shows the correspondence between function approximation method and derivative approximation method for 11172 test cases provided by the authors of LLDRLF. The table shows, for instance, that the 3033 cases where the Bratio Method was chosen to compute the function value were distributed over all five of the derivative computing methods, with the majority of cases being handled by the Direct Differentiation Method. In contrast, ADIFOR never modifies the control flow of the original function code; the ADIFOR-generated derivatives are based entirely upon the original partitioning of the input domain and the original approximation methods.

[1]This work was supported by the National Aerospace Agency under Cooperative Agreement No. NCCW-0027 and by the National Science Foundation, through the Center for Research on Parallel Computation, under Cooperative Agreement No. CCR-9120008.

Function Method	Derivative Method: Asymptotic	Continued Fraction	Direct Diff.	Extreme Value	Series
Asymptotic	153	0	153	0	0
Bratio	435	551	1567	375	105
Continued Fraction	0	805	805	0	0
Extreme Value	0	0	2541	3100	0
Series	0	0	291	230	61

Table 1 Correspondence of Function and Derivative Methods on 11172 Test Cases

It should be noted, that in most applications of ADIFOR, hand-coded derivatives are not available for use in accuracy studies. The use of finite-difference approximations is usually worthwhile as a means of checking the results of automatic differentiation.[2] In the case of LLDRLF, we were able to find finite-difference stepsizes that gave derivative approximations that agreed well with the hand-coded derivatives.

The ADIFOR-generated derivative code and the hand-coded derivative code were both run on the 11172 test cases. Table 2 shows the results of the initial comparison using double and quad precision for all arithmetic operations. Test cases listed under 0 digits accuracy include cases in which IEEE exceptional values, such as `NAN` or `INF`, were generated.

Digits Accuracy	0	1	2	3	4	5	6	7	8	9	10	11	12..15
1st Derivs (Double Precision)	16	0	0	0	0	0	0	0	0	15	75	253	10813
2nd Derivs (Double Precision)	236	15	23	26	25	8	17	11	10	33	116	325	10327
1st Derivs (Quad Precision)	10	0	0	0	0	0	0	0	0	8	6	127	11021
2nd Derivs (Quad Precision)	236	5	4	4	4	2	5	2	5	23	33	161	10688

Table 2 Accuracy Comparison (Initial)

The ADIFOR-generated code monitors the values passed to intrinsic functions to determine if derivatives are being computed at exceptional points. For example, any time the derivative of a square root is required, ADIFOR also generates code to check to see if the argument to the derivative of the square root is 0.0. If so, then the run-time system prints an error message. For the 11172 test cases, the exception handler reported that `SQRT` and `ABS` were being differentiated at the 0.0 value for a small number of the test cases. In each of these test cases, the final derivative result had fewer than 6 digits of agreement with the hand-coded derivatives.

2 Analysis of Discrepancies

There are two classes of discrepancies that occur in the computation of derivatives by automatic differentiation:

Expression An expression discrepancy results from the way in which a value is computed. For example, $y = \sqrt{x}\sqrt{x}$ has derivatives everywhere (since it can also be written as $y = x$), but attempting to differentiate the *expression* $y = \sqrt{x}\sqrt{x}$ via the chain rule will result in an expression that is undefined at 0. In the ADIFOR derivatives for LLDRLF, the SQRT function generates an expression discrepancy.

Numerical A numerical discrepancy results when the derivative code overflows or underflows, even though the original function code did not.

[2] In the vast majority of cases, however, when finite differences and ADIFOR-generated derivatives disagree, the ADIFOR-generated derivative are correct.

As we examined the discrepancies in the ADIFOR-generated code for LLDRLF, we investigated methods and techniques that could be used to reduce the effects of the discrepancies. We constrained our choice of techniques, for paradigmatic reasons, to include only those that could be applied at the level of the *original source* code. Rewriting of the generated code violates the purpose of automatic differentiation. Hence, any alteration of the generated code must be a change that can be made at the original source level.

2.1 Expression Discrepancies

We began by examining the cause of the `SQRT` and `ABS` exception handler reports. We found that the `SQRT` reports correspond to the expression discrepancy shown in Figure 1. When `lambda` has the value `0.0`, `f` is assigned the value `0.0`, which leads to an exception at the line with label 30.

```
      ...
   20 CALL rlog1(-lambda/a,temp1)
      CALL rlog1(lambda/b,temp2)
      f = a*temp1 + b*temp2
      t = exp(-f)
      IF (t.EQ.0.0D0) RETURN
   30 z0 = sqrt(f)
      ...
```

Figure 1 An Expression Discrepancy

We were able to ascertain that the variable `lambda` receives the value `0.0` or a value extremely close to `0.0` whenever the input `w` to LLDRLF is `0.0`. The LLDRLF function (mathematically speaking) has continuous 1st and 2nd derivatives, hence, we can treat the anomaly when `w` is `0.0` as a *removable* discontinuity. To do so, we chose to perturb the value of `w` by 10^{-12} whenever a `w` value of `0.0` was encountered.

Digits Accuracy	0	1	2	3	4	5	6	7	8	9	10	11	12..15
1st Derivs (Double Precision)	8	0	0	0	0	0	0	4	12	35	103	289	10721
2nd Derivs (Double Precision)	228	3	7	7	19	27	41	23	16	33	116	351	10301
1st Derivs (Quad Precision)	8	0	0	0	0	0	0	4	12	28	34	158	10928
2nd Derivs (Quad Precision)	228	0	0	0	0	0	1	1	9	37	54	191	10651

Table 3 Accuracy Comparison (After `w` Perturbation)

Table 3 shows the comparison of results generated by the hand-coded derivative and the ADIFOR-generated code on the same 11172 test cases, as above, but with the value of `w` perturbed as just described for the ADIFOR-generated code.

```
      if (w .eq. 0.0) w = w + 1.0e-12
```

Figure 2 The Expression Discrepancy Patch

The code shown in Figure 2 presents a one line patch that has the desired effect of setting `w` to 10^{-12} without modifying the value of `g_w`. Simply assigning the value `0.0` to `w` would have had the effect of setting `g_w` to `0.0`, as well, and causing all of the ADIFOR-generated derivatives to have the value `0.0`. Perturbing the value of `w` also served to remove the `ABS` exception handler reports.

2.2 Numerical Discrepancies

The computation of IEEE `NAN` and `INF` values indicated the presence of possible numerical discrepancies in the ADIFOR-generated code for LLDRLF. To determine where the `INF` and `NAN` values were first being computed, we reapplied ADIFOR to LLDRLF with the `AD_TRACE_RESULTS` flag set to `true` to force the values of intermediate derivative values to be monitored. Whenever an ADIFOR-generated statement assigned a

```
      ...
      CALL bratio(b,a,x,y,ltail,lcum,ierr)
      qtail = ltail .LE. lcum
      IF (min(ltail,lcum).GT.zero) THEN
          iwhich = 2
          IF (qtail) THEN
              ltail = log(ltail)
              CALL ctc(ltail,lcum)
          ELSE
              lcum = log(lcum)
              CALL ctc(lcum,ltail)
          END IF

          RETURN

      END IF
      ...
```

Figure 3 Source of Numerical Discrepancy

```
        ...
        if (min(ltail, lcum) .gt. zero) then
          iwhich = 2
          if (qtail) then
            ...
            h_g_ltail = (1.0d0/ltail) * h_g_ltail +
     +                  (-1.0d0/(ltail * ltail)) * g_ltail * h_ltail
            ...
          else
            ...
            h_g_lcum = (1.0d0/lcum) * h_g_lcum +
     +                 (-1.0d0/(lcum * lcum)) * g_lcum * h_lcum
            ...
          endif
          return
        endif
```

Figure 4 Derivative Code for Numerical Discrepancy

value of `NAN` or `INF`, the line number and file name for that statement was logged to an output file. Executing the tracing version of the ADIFOR-generated LLDRLF code indicated that all of the `NAN` and `INF` values were being generated while computing first and second derivatives of the intermediate variables `ltail` and `lcum` as shown in Figure 3 using the code shown in Figure 4.

As an example of how the `NAN` and `INF` values were generated, consider one test case in which `h_g_bratio` computed 10^{-310} as the value of `ltail`. When computing the value of `h_g_ltail`, in either double or quad precision, with an `ltail` of 10^{-310}, $1.0/$`ltail` evaluates to `INF`, $-1.0/($`ltail` $*$ `ltail`$)$ evaluates to `-INF`, and `h_g_ltail` evaluates to `NAN`.

By reassociating the computations for `h_g_ltail` and `h_g_lcum`, as shown in Figure 5, `NAN` and `INF` values can be avoided for each of the test cases. Table 4 compares the results of the ADIFOR-generated code after both perturbing `w` and reassociating the calculation of `h_g_ltail` and `h_g_lcum`. Notice that in both double and quad precision, there still remains a single test case in which the ADIFOR-generated code fails to provide any digits of accuracy in the computed derivative.

We explored one other technique for avoiding the generation of `NAN` and `INF` values. As shown in Figure 3, LLDRLF applies the Bratio Method, and then checks the values of `ltail` and `lcum` to see if they are both greater than `0.0`. If one of the values is not greater than `0.0`, the Asymptotic Method is applied. In double precision, the `NAN` and `INF` values are generated when `ltail` or `lcum` have a value that is smaller than about 10^{-150}. Simply changing the acceptance test for the Bratio Method as shown in Figure 6 in the LLDRLF code, gives very satisfactory derivative results as shown in Table 5. In essence, we have achieved a minor repartioning of the input domain with a very local source code modification.

```
      ...
      if (min(ltail, lcum) .gt. zero) then
        iwhich = 2
        if (qtail) then
          ...
          h_g_ltail = h_g_ltail/ltail -
     +                (g_ltail/ltail) * (h_ltail/ltail)
          ...
        else
          ...
          h_g_lcum = h_g_lcum/lcum -
     +               (g_lcum/lcum) * (h_lcum/lcum)
          ...
        endif
        return
      endif
```

Figure 5 Reassociated Derivative Code for Numerical Discrepancy

Digits Accuracy	0	1	2	3	4	5	6	7	8	9	10	11	12..15
1st Derivs (Double Precision)	0	0	0	1	0	0	0	4	4	12	35	103	10551
2nd Derivs (Double Precision)	1	3	6	12	14	27	43	23	23	16	45	139	9758
1st Derivs (Quad Precision)	0	0	0	1	0	0	0	4	12	28	34	159	10934
2nd Derivs (Quad Precision)	1	0	0	0	0	0	1	2	10	46	57	216	10839

Table 4 Accuracy Comparison (After `w` Perturbation and Reassociation)

3 Summary

We note that naive usage of ADIFOR resulted in first derivatives that were accurate to 6 figures in 99.9% of the tested cases, and second derivatives that were accurate in 97.0% of the cases, in double precision. In quad precision, first derivatives were accurate to 6 figures in 99.9% of the tested cases, and second derivatives were accurate in 97.7% of the cases. After making two one-line changes to the LLDRLF source code we achieved double precision first derivatives and quad precision first and second derivatives accurate to 6 figures in 100% of the tested cases, and double precision second derivatives that were accurate in 99.4% of the cases. In all of the cases where the ADIFOR-generated second derivatives had fewer than 6 figures of accuracy, the ADIFOR-generated code provided at least 1 digit of accuracy, and averaged 3.9 digits of accuracy.

It is encouraging that the presence of all of the discrepancies between the hand-coded derivatives and the ADIFOR-generated derivatives could be determined by looking at the output of the ADIFOR-generated code – `SQRT` and `ABS` exception handler messages were generated for the expression discrepancies, and IEEE `NAN` and `INF` values were returned for the numerical discrepancies. In addition, by invoking ADIFOR with the "AD_TRACE_RESULTS" option enabled, it was easy to detect where each of the numerical discrepancies occurred in the code.

```
      ...
      CALL bratio(b,a,x,y,ltail,lcum,ierr)
      qtail = ltail .LE. lcum
      IF (min(ltail,lcum).GT.1.0d-150) THEN
        ...
      END IF
      ...
```

Figure 6 Repartitioned Derivative Code for Numerical Discrepancy

Digits Accuracy	0	1	2	3	4	5	6	7	8	9	10	11	12..15
1st Derivs (Double Precision)	0	0	0	0	0	0	0	4	12	34	106	280	10736
2nd Derivs (Double Precision)	0	3	7	7	19	27	41	23	16	44	132	362	10491
1st Derivs (Quad Precision)	0	0	0	0	0	0	0	4	12	28	34	158	10936
2nd Derivs (Quad Precision)	0	0	0	0	0	0	1	1	9	45	56	215	10845

Table 5 Accuracy Comparison (After `w` Perturbation and Repartitioning)

Based on the results of this study, users of ADIFOR are advised to take a close look at any messages generated by ADIFOR's exception handler. The cause of IEEE exceptions which appear when the ADIFOR-generated code is executed, but which did not appear in the original function evaluation, should also be examined carefully.

This study also appears to indicate that ADIFOR should probably be more careful about how it generates derivative code for `log`. Unfortunately, to maximize the accuracy of derivative computations, ADIFOR would need to generate several different versions of the derivative code and then select the best version to execute based on the value of the argument to `log` and the values of the derivatives of the argument with respect to the independent variables.

4 Acknowledgements

We thank Barry Brown and Kathy Russell at the University of Texas, M.D. Anderson Cancer Center, for providing us with LLDRLF, the hand-generated derivative code for LLDRLF, and the program that compared the results from the hand-generated and ADIFOR-generated derivative code.

References

[1] C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–31, 1992. Also available as ADIFOR Working Note #1, Technical Report MCS-P263-0991, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR91185, Center for Research on Parallel Computation, Rice University.

[2] C. Bischof, A. Carle, P. Khademi, and A. Mauer. The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs, 1994. Technical Report MCS-P381-1194, Mathematics and Computer Science Division, Argonne National Laboratory, and CRPC-TR94491, Center for Research on Parallel Computation, Rice University.

[3] Barry W. Brown, F. Martin Spears, Lawrence B. Levy, James Lovato, and Kathy Russell. LLDRLF, Log-likelihood and some derivatives for Log-F models. Submitted to Transactions on Mathematical Software, Feb 1994. Available by ftp at `odin.mda.uth.tmc.edu`, `pub/accflf`.

[4] Armido R. Didonato and Alfred H. Morris, Jr. Algorithm 708: Significant digit computation of the incomplete beta function ratios. *ACM Transactions on Mathematical Software*, 18(3):360–373, Sept 1992.