AD *in* Fortran
Part 1: Design

Alexey Radul, Barak A. Pearlmutter, and Jeffrey Mark Siskind

Abstract We propose extensions to FORTRAN which integrate forward and reverse Automatic Differentiation (AD) directly into the programming model. Irrespective of implementation technology, embedding AD constructs directly into the language extends the reach and convenience of AD while allowing abstraction of concepts of interest to scientific-computing practice, such as root finding, optimization, and finding equilibria of continuous games. Multiple different subprograms for these tasks can share common interfaces, regardless of whether and how they use AD internally. A programmer can maximize a function `F` by calling a library maximizer, `XSTAR=ARGMAX(F, X0)`, which internally constructs derivatives of `F` by AD, without having to learn how to use any particular AD tool. We illustrate the utility of these extensions by example: programs become much more concise and closer to traditional mathematical notation. A companion paper describes how these extensions can be implemented by a program that generates input to existing FORTRAN-based AD tools.

Key words: Nesting, multiple transformation, forward mode, reverse mode, TAPENADE, ADIFOR, programming-language design

1 Introduction

The invention of FORTRAN was a major advance for numeric computing, allowing

Alexey Radul
Hamilton Institute, National University of Ireland Maynooth, Ireland, aradul@nuim.ie

Barak A. Pearlmutter
Hamilton Institute, National University of Ireland Maynooth, Ireland, barak@cs.nuim.ie

Jeffrey Mark Siskind
Electrical and Computer Engineering, Purdue University, IN, USA, qobi@purdue.edu

$$g(x;\alpha,\beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

to be transcribed into a natural but unambiguous notation

```
      FUNCTION G(X,ALPHA,BETA)
      G=BETA**ALPHA/GAMMA(ALPHA)*X**(ALPHA-1)*EXP(-BETA*X)
      END
```

which could be automatically translated into an executable program. However, transcribing

$$x_{i+1} = x_i - f(x_i)/f'(x_i)$$

to FORTRAN in

```
      FUNCTION RAPHSN(F, FPRIME, X0, N)
      EXTERNAL F, FPRIME
      X = X0
      DO 1690 I=1,N
 1690 X = X-F(X)/FPRIME(X)
      RAPHSN = X
      END
```

requires that the *caller* provide both `F` and `FPRIME`. Manually coding the latter from the former is, in most cases, a mechanical process, but tedious and error prone.

This problem has traditionally been addressed by arranging for an AD preprocessor to produce `FPRIME` [12, 14]. That breakthrough technology not only relieves the programmer of the burden of mechanical coding of derivative-calculation codes, it also allows the derivative code to be updated automatically, ensuring consistency and correctness. However, this *caller derives* discipline has several practical difficulties. First, the user must learn how to use the AD preprocessor, which constitutes a surprisingly serious barrier to adoption. Second, it makes it very difficult to experiment with the use of different sorts of derivatives (e.g., adding a Hessian-vector product step in an optimization) in such called subprograms, or to experiment with different AD preprocessors. Third, although preprocessors might be able to process code which has already been processed in order to implement nested derivatives, the maneuvers required by current tools can be somewhat arcane [10]. Fourth, software engineering principles of locality and atomicity are being violated: knowledge of what derivatives are needed is distributed in a number of locations which must be kept consistent; and redundant information, which must also be kept consistent, is being passed, often down a long call chain. We attempt to solve these problems, making the use of AD more concise, convenient, and intuitive to the scientific programmer, while keeping to the spirit of FORTRAN. This is done using the *Forward And Reverse Fortran Extension Language* or FARFEL, a small set of extensions to FORTRAN, in concert with an implementation strategy which leverages existing FORTRAN compilers and AD preprocessors [2, 5].

The remainder of the paper is organized as follows: Section 2 describes FARFEL. Section 3 describes a concrete example FARFEL program to both motivate and illuminate the proposed extensions. Section 4 situates this work in its broader context.

Section 5 summarizes this work's contributions. A companion paper [9] describes how Farfel can be implemented by generating input to existing AD tools.

2 Language Extensions

Farfel consists of two principal extensions to Fortran: syntax for AD and for nested subprograms. We currently support only Fortran77, but there is no barrier, in principle, to adding Farfel to more recent dialects.

Extension 1: AD Syntax
Traditional mathematical notation allows one to specify

$$\phi' = \frac{\mathrm{d}}{\mathrm{d}\sigma}\left(\frac{1}{\sqrt{2\pi\sigma^2}}\exp-\frac{1}{2}\left(\frac{x-\bar{x}}{\sigma}\right)^2\right)$$

By analogy, we extend Fortran to encode this as

```
      ADF(TANGENT(SIGMA) = 1)
      PHI = 1/SQRT(2*PI*SIGMA**2)*EXP(-0.5*((X-XBAR)/SIGMA)**2)
      END ADF(PHIPRM = TANGENT(PHI))
```

which computes the derivative `PHIPRM` of `PHI` with respect to `SIGMA` by forward AD. For syntactic details see companion paper [9].

An analogous Farfel construct supports computing the same derivative with reverse AD:

```
      ADR(COTANGENT(PHI) = 1)
      PHI = 1/SQRT(2*PI*SIGMA**2)*EXP(-0.5*((X-XBAR)/SIGMA)**2)
      END ADR(PHIPRM = COTANGENT(SIGMA))
```

Note that with the **`ADR`** construct, the *dependent* variable appears at the beginning of the block and the *independent* variable at the end—the variables and assignments in the opening and closing statements specify the desired inputs to and outputs from the reverse phase, whereas the statements inside the block give the forward phase.

These constructs allow not just convenient expression of AD, but also modularity and encapsulation of code which employs AD. For instance, we can write a general scalar-derivative subprogram `DERIV1` at user level

```
      FUNCTION DERIV1(F, X)
      EXTERNAL F
      ADF(X)
      Y = F(X)
      END ADF(DERIV1 = TANGENT(Y))
      END
```

which could be used in, for example,

```
      FUNCTION PHI(SIGMA)
      PHI = 1/SQRT(2*PI*SIGMA**2)*EXP(-0.5*((X-XBAR)/SIGMA)**2)
      END
      PHIPRM = DERIV1(PHI, SIGMA)
```

DERIV1 can be changed to use reverse AD without changing its API:

```
      FUNCTION DERIV1(F, X)
      EXTERNAL F
      ADR(Y)
      Y = F(X)
      END ADR(DERIV1 = COTANGENT(X))
      END
```

allowing codes written with DERIV1 to readily switch between using forward and reverse AD.

To take a more elaborate example, we can write a general gradient calculation GRAD using repeated forward AD:

```
      SUBROUTINE GRAD(F, X, N, DX)
      EXTERNAL F
      DO 1492 I=1,N
      ADF(TANGENT(X(J)) = 1-MIN0(IABS(I-J),1), J=1,N)
      Y = F(X)
 1492 END ADF(DX(I) = TANGENT(Y))
      END
```

(Note that the **ADF** and **ADR** constructs support implied-**DO** syntax for arrays.)

This can be modified to instead use reverse AD without changing the API:

```
      SUBROUTINE GRAD(F, X, N, DX)
      EXTERNAL F
      ADR(Y)
      Y = F(X)
      END ADR(DX(I) = COTANGENT(X(I)), I=1,N)
      END
```

Although not intended to support checkpoint-reverse AD, our constructs are sufficiently powerful to express a reverse checkpoint:

```
C       CHECKPOINT REVERSE F->G.  BOTH 1ST ARG IN, 2ND ARG OUT
       CALL F(X, Y)
       ADR(COTANGENT(Z(I)) = ..., I=1,NZ)
       CALL G(Y, Z)
       END ADR(DY(I) = COTANGENT(Y(I)), I=1,NY)
       ADR(COTANGENT(Y(I)) = DY(I), I=1,NY)
       CALL F(X, Y)
       END ADR(DX(I) = COTANGENT(X(I)), I=1,NX)
```

This sort of encapsulation empowers numeric programmers to conveniently experiment with the choice of differentiation method, or with the use of various sorts of derivatives, including higher-order derivatives, without tedious modification of lengthy call chains.

Extension 2: Nested Subprograms

We borrow from Algol 60 [1] and generalize the Fortran “statement function” construct by allowing subprograms to be defined inside other subprograms, with lexical scope.

For example, given a univariate maximizer ARGMAX, we can express the idea of line search as follows:

```
C     MAXIMIZE F ALONG THE LINE PARALLEL TO XDIR THROUGH X
      SUBROUTINE LINMAX(F, X, XDIR, LENX, N, XOUT)
      EXTERNAL F
      DIMENSION Y(50)
        FUNCTION ALINE(DIST)
        DO 2012 I=1,LENX
 2012   Y(I) = X(I)+DIST*XDIR(I)
        ALINE = F(Y)
        END
      BESTD = ARGMAX(ALINE, 0.0, N)
      DO 2013 I=1,LENX
 2013 XOUT(I) = X(I)+BESTD*XDIR(I)
      END
```

Here we are using a library univariate maximizer to maximize the univariate function `ALINE`, which maps the distance along the given direction to the value of our multidimensional function of interest `F` at that point. Note that `ALINE` refers to variables defined in its enclosing scope, namely `F`, `X`, `XDIR`, `LENX`, and `Y`. Note that if `ARGMAX` uses derivative information, AD will be performed automatically on `ALINE`.

3 Concrete Example

We employ a concrete example to show the convenience of the above constructs. We will also illustrate the implementation on this example in [9]. Let two companies, Apple and Banana, be engaged in competition in a common fashion accessories market. Each chooses a quantity of their respective good to produce, and sells all produced units at a price determined by consumer demand. Let us model the goods as being distinct, but partial substitutes, so that availability of products of A decreases demand for products of B and vice versa (though perhaps not the same amount). We model both companies as having market power, so the price each gets will depend on both their own output and their competitor's. Each company faces (different) production costs and seeks to maximize its profit, so we can model this situation as a two player game. Let us further assume that the quantities involved are large enough that discretization effects can be disregarded.

An equilibrium (a^*, b^*) of a two-player game with continuous scalar strategies a and b and payoff functions $A(a,b)$ and $B(a,b)$ must satisfy a system of equations:

$$a^* = \operatorname*{argmax}_a A(a, b^*) \qquad\qquad b^* = \operatorname*{argmax}_b B(a^*, b) \tag{1}$$

Equilibria can be sought by finding roots of

$$a^* = \operatorname*{argmax}_a A(a, \operatorname*{argmax}_b B(a^*, b)) \tag{2}$$

which is the technique we shall employ.[1] Translated into computer code in the most natural way, solving this equation involves a call to an optimization subprogram within the function passed to an optimization subprogram, itself within the function passed to a root-finding subprogram. If said optimization and root-finding subprograms need derivative information, this gives rise to deeply nested AD.

Note that in (2), the payoff function B is bivariate but argmax takes a univariate (in the variable of maximization) objective function. The a^* variable passed to B is *free* in the innermost argmax expression. Free variables occur naturally in mathematical notation, and we support them by allowing nested subprogram definitions.

We can use our extensions to code finding the roots of (2) in a natural style:

```
C     ASTAR & BSTAR: GUESSES IN, OPTIMIZED VALUES OUT
      SUBROUTINE EQLBRM(BIGA, BIGB, ASTAR, BSTAR, N)
      EXTERNAL BIGA, BIGB
        FUNCTION F(ASTAR)
          FUNCTION G(A)
            FUNCTION H(B)
            H = BIGB(ASTAR, B)
            END
          BSTAR = ARGMAX(H, BSTAR, N)
          G = BIGA(A, BSTAR)
          END
        F = ARGMAX(G, ASTAR, N)-ASTAR
        END
      ASTAR = ROOT(F, ASTAR, N)
      END
```

where we implement just the minimal cores of one-dimensional optimization and root finding to illustrate the essential point — root finding by the Rhapson method:

```
      FUNCTION ROOT(F, X0, N)
      X = X0
      DO 1669 I=1,N
      CALL DERIV2(F, X, Y, YPRIME)
 1669 X = X-Y/YPRIME
      ROOT = X
      END
```

```
      SUBROUTINE DERIV2(F, X, Y, YPRIME)
      EXTERNAL F
      ADF(X)
      Y = F(X)
      END ADF(YPRIME = TANGENT(Y))
      END
```

and optimization by finding the root of the derivative:

```
      FUNCTION ARGMAX(F, X0, N)
        FUNCTION FPRIME(X)
        FPRIME = DERIV1(F, X)
        END
      ARGMAX = ROOT(FPRIME, X0, N)
      END
```

[1] The existence or uniqueness of an equilibrium is not in general guaranteed, but our particular A and B have a unique equilibrium. Coordinate descent (alternating optimization of a^* and b^*) would require less nesting, but has inferior convergence properties. Although this example involves AD through iterative processes, we do not address that issue in this work: it is beyond the scope of this paper, and used here only in a benign fashion, for vividness.

On our concrete objective functions these converge rapidly, so for clarity we skip the clutter of convergence detection.

This strategy impels us to compute derivatives nested five deep, in a more complicated pattern than just a fifth-order derivative of a single function. This undertaking is nontrivial with current AD tools [10], but becomes straightforward with the proposed extensions—embedded AD syntax and nested subprograms make it straightforward to code sophisticated methods that require complex patterns of derivative information.

4 Discussion

The FARFEL AD extensions hew to the spirit of FORTRAN, which tends to prefer blocks rather than higher-order operators for semantic constructs. The most straightforward implementation technology involves changing transformed blocks into subprograms that capture their lexical variable context and closure-converting these into top-level subprograms, rendering them amenable to processing with existing tools [9]. Since the machinery for nested subprograms is present, allowing them imposes little additional implementation effort. Moreover, as seen in the example above, that extension makes code that involves heavy use of higher-order functions, which is encouraged by the availability of the AD constructs, more straightforward. In this sense AD blocks and nested subprograms interact synergistically.

These new constructs are quite expressive, but this very expressiveness can tax many implementations, which might not support some combinations or usages. For instance, code which makes resolution of the AD at compile time impossible (an *n*-th derivative subprogram, say) would be impossible to support without a dynamic run-time AD mechanism. This would typically not be available. Another common restriction would be that many tools do not support reverse mode at all and even those that do typically do not allow nesting over reverse mode, either reverse-over-reverse or forward-over-reverse. It is the responsibility of the implementation to reject such programs with a cogent error.

The FARFEL extensions are implemented by the FARFALLEN preprocessor [9], which generates input for and invokes existing AD tools. This leverages existing AD systems to provide the differentiation functionality in a uniform and integrated way, extending the reach of AD by making its use easier, more natural, and more widely applicable.

Such a prepreprocessor can target different AD systems (like ADIFOR [2] and TAPENADE [5]), allowing easy porting of code from one AD system to another. It could even mix AD systems, for example using TAPENADE to reverse-transform code generated by using ADIFOR in forward mode, capturing their respective advantages for the application at hand. The effort of implementing such retargetings and mixings could then be factored to one developer (of the prepreprocessor) instead of many end users of AD.

A more important benefit of extending FORTRAN with AD syntax and nested subprograms is that a host of notions become reusable abstractions—not just first-order derivatives, but also their variations, combinations, and uses, e.g., Jacobians, Hessians, Hessian-vector products, filters, edge detectors, Fourier transforms, convolutions, Hamiltonians, optimizations, integration, differential-equation solvers. The interfaces to different methods for these tasks can be made much more uniform because, as our `ARGMAX` did, they can accept subprograms that just accept the variables of interest (in this case, the argument of maximization) and take any needed side information from their lexical scope; and subprograms such as `ARGMAX` can obtain any derivative information they wish from AD without having to demand it be passed in as arguments. So different maximization methods can be tried out on the same objective function with ease, regardless of how much derivative information they require; and at the same time, different objective functions, that carry different side information, can be maximized by the same maximization subprogram without having to adjust it to transmit the needed side information. Essentially, derivatives are requested where they are needed, and the implementation does the necessary bookkeeping.

These modularity benefits are illustrated by our example program: the FARFEL input is only 63 lines of code, whereas the amount of code it expands into, which is comparable to what would need to be written by hand without these extensions, weighs in at a much more substantial 164 for TAPENADE and 315 for ADIFOR, including the configuration data needed to run the AD preprocessors to produce the needed derivatives. Manually performing the 5 nested applications of AD this example calls for is a tedious, error prone, multi-hour effort, which must be undertaken separately for each preprocessor one wishes to try.[2] Existing AD tools do already save the major labor of manually implementing derivative and gradient subprograms, and keeping them in sync with the subprograms being differentiated. The further preprocessing step outlined above leverages these tools into being even more useful. For larger programs, the savings of implementation and maintenance effort would be considerable.

The present suggestion is not, of course, limited to FORTRAN. Nested subprograms have gained wide adoption in programming-language designs from ALGOL 60 and beyond, and have yielded proven gains in programmer productivity. Their advantages for code expressiveness have led to functions with lexical scope being used as a mathematical formalism for reasoning about computing [4], to programming languages organized around the function as the primary program construct [7], and to compilers that specialize in the efficient representation and use of functions [6, 13].

One can also add **`ADF`**- and **`ADR`**-like constructs to other languages that have preprocessor implementations of AD, for example, C and ADIC [3]. One would not even need to add nested subprograms in the preprocessor, because GCC already implements them for GNU C. Doing so would expand the convenience (and therefore reach) of existing AD technology even further.

[2] A detailed step-by-step discussion of the transformation of this example along with all intermediate code is available at `http://www.bcl.hamilton.ie/~qobi/fortran/`.

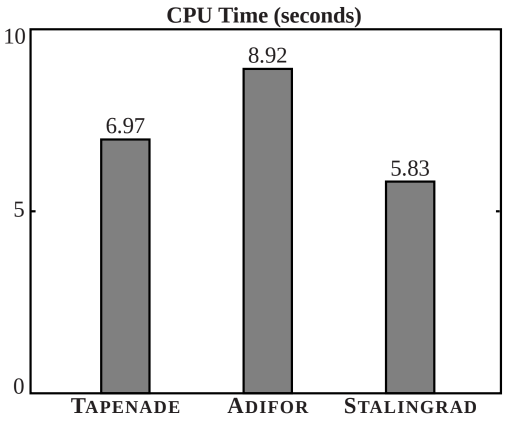

Fig. 1 Performance comparison. Smaller is faster. Numeric solution of (2) with above FARFEL code, $N = 1000$ iterations at each level, FARFALLEN targeting two FORTRAN-based AD tools; for comparison, the same computation is coded in VLAD [8] and compiled with STALINGRAD [11]. Computer: Intel i7 870 @ 2.93GHz, GFORTRAN 4.6.2-9, 64-bit Debian sid, `-Ofast -fwhole-program`, single precision. See [9] for details.

Retrofitting AD onto existing languages by preprocessing is not without its limitations, however. Efficient AD preprocessors must construct a call graph in order to determine which subprograms to transform, along with a variety of other tasks already performed by the compiler. Moreover, optimizing compilers cannot be relied upon to aggressively optimize intricate machine-generated code, as such code often exceeds heuristic cutoffs in various optimization transformations. This imposes a surprisingly serious limitation on AD preprocessors. (Together, these also imply a significant duplication of effort, while providing room for semantic disagreements between AD preprocessors and compilers which can lead to subtle bugs.) This leads us to anticipate considerable performance gains from designing an optimizing compiler with integrated AD. Indeed, translating our concrete example into VLAD [8] and compiling with STALINGRAD [11], our prototype AD-enabled compiler, justifies that suspicion (see Fig. 1). We therefore plan to make a VLAD back-end available in version 2 of FARFALLEN.

5 Conclusion

We have defined and motivated extensions to FORTRAN for convenient, modular programming using automatic differentiation. The extensions can be implemented as a prepreprocessor [9]. This strategy enables modular, flexible use of AD in the context of an existing legacy language and tool chain, without sacrificing the desirable performance characteristics of these tools: only about 20%–50% slower than a dedicated AD-enabled compiler, depending on which FORTRAN AD system is used.

Acknowledgements This work was supported, in part, by Science Foundation Ireland grant 09/IN.1/I2637, National Science Foundation grant CCF-0438806, the Naval Research Laboratory under Contract Number N00173-10-1-G023, and the Army Research Laboratory accomplished under Cooperative Agreement Number W911NF-10-2-0060. Any views, opinions, findings, conclusions, or recommendations contained or expressed in this document or material are those of the author(s) and do not necessarily reflect or represent the views or official policies, either expressed or implied, of SFI, NSF, NRL, the Office of Naval Research, ARL, or the Irish or U.S. Governments. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation herein.

References

1. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language ALGOL 60. The Computer Journal **5**(4), 349–367 (1963). DOI 10.1093/comjnl/5.4.349
2. Bischof, C.H., Carle, A., Corliss, G.F., Griewank, A., Hovland, P.D.: ADIFOR: Generating derivative codes from Fortran programs. Scientific Programming **1**(1), 11–29 (1992)
3. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience **27**(12), 1427–1456 (1997). DOI 10.1002/(SICI)1097-024X(199712)27:12⟨1427::AID-SPE138⟩3.0.CO;2-Q. URL `http://www-fp.mcs.anl.gov/division/software`
4. Church, A.: The Calculi of Lambda Conversion. Princeton University Press, Princeton, NJ (1941)
5. Hascoët, L., Pascual, V.: TAPENADE 2.1 user's guide. Rapport technique 300, INRIA, Sophia Antipolis (2004). URL `http://www.inria.fr/rrrt/rt-0300.html`
6. Jones, S., Hall, C., Hammond, K., Partain, W., Wadler, P.: The Glasgow Haskell compiler: a technical overview. In: Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference, vol. 93 (1993)
7. Jones, S.P.: Haskell 98 language and libraries: the revised report. Journal of Functional Programming **13**(1) (2003)
8. Pearlmutter, B.A., Siskind, J.M.: Using programming language theory to make automatic differentiation sound and efficient. In: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (eds.) Advances in Automatic Differentiation, *Lecture Notes in Computational Science and Engineering*, vol. 64, pp. 79–90. Springer, Berlin (2008). DOI 10.1007/978-3-540-68942-3_8
9. Radul, A., Pearlmutter, B.A., Siskind, J.M.: AD in Fortran, Part 2: Implementation. In: Advances in Automatic Differentiation, Lecture Notes in Computational Science and Engineering. Springer, Berlin (2012)
10. Siskind, J.M., Pearlmutter, B.A.: Putting the automatic back into AD: Part I, What's wrong. Tech. Rep. TR-ECE-08-02, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA (2008). URL `http://docs.lib.purdue.edu/ecetr/368`
11. Siskind, J.M., Pearlmutter, B.A.: Using polyvariant union-free flow analysis to compile a higher-order functional-programming language with a first-class derivative operator to efficient Fortran-like code. Tech. Rep. TR-ECE-08-01, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA (2008). URL `http://docs.lib.purdue.edu/ecetr/367`
12. Speelpenning, B.: Compiling fast partial derivatives of functions given by algorithms. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL (1980)
13. Steele Jr., G.L.: RABBIT, a compiler for Scheme. Tech. Rep. TR474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1978)
14. Wengert, R.: A simple automatic derivative evaluation program. Communications of the ACM **7**(8), 463–464 (1964)