ANL/MCS-TM-310

Automatic Differentiation of Codes in Nuclear Engineering Applications

MIHAI ALEXE
Computational Science Laboratory, Virginia Polytechnic Institute and State University
mihai@vt.edu

OLEG RODERICK, JEAN UTKE, MIHAI ANITESCU [*], PAUL HOVLAND
Mathematics and Computer Science Division, Argonne National Laboratory
{roderick,utke,hovland,anitescu}@mcs.anl.gov

THOMAS FANNING
Nuclear Engineering Division, Argonne National Laboratory
fanning@anl.gov

December 8, 2009

[*]Corresponding author *anitescu@mcs.anl.gov*

Contents

Abstract

We discuss our experience in applying automatic differentiation (AD) to calculations in nuclear reactor applications. The document is intended as a guideline on how to apply AD to Fortran codes with significant legacy components; it is also a part of a larger research effort in uncertainty quantification using sampling methods augmented with derivative information. We provide a brief theoretical description of the concept of AD,explain the necessary changes in the code structure, and remark on possible ways to deal with non-differentiability. Numerical experiments were carried out where the derivative of a functional subset of the SAS4A/SASSYS code was computed in forward mode with several AD tools. The results are in good agreement with both the real and complex finite-difference approximations of the derivative.

Keywords Automatic Differentiation, Fortran 77, Nuclear Engineering, Derivative, Gradient

AMS Subject Classification 58C20,56G05

1 Introduction

In this report, we discuss our ongoing work in applying automatic differentiation (AD) [14] to calculations with the nuclear reactor simulation code MATWS. The MATWS computer program has been compiled as part of an application library. It combines the point-kinetics module from the SAS4A/SASSYS computer code with a simplified representation of a reactor heat-removal system. A description of the underlying mathematical model is available, but the complete process of obtaining the numerical solution is documented only in the code comments, and sparsely at that. MATWS was used, in combination with another simulation tool, GoldSim, to model unprotected loss-of-heat-sink, loss-of-flow and transient-overpower accidents [21].

This document is intended as a guideline on how to apply AD to Fortran codes that contain legacy components. It is also part of a larger research effort in uncertainty quantification using sampling methods augmented with derivative information [23, 24]. We provide a brief theoretical description of the concept of AD, note what AD software is available, explain the necessary changes in the code structure, and remark on possible ways to deal with nondifferentiability.

To date, we have resolved the basic issues of compatibility of the code with AD tools. We have obtained derivative values using different AD tools that agree with each other, with the estimates obtained by finite differences, and with the results given by the complex differentiation approach (a technique closely related to forward-mode differentiation [20]).

Ideally, human participation in the automatic differentiation process should be limited to identifying the dependent and independent variables of interest and to debugging the code that produces derivatives, at its compilation stage. In practice, we cannot claim this smooth process as an operational norm, nor claim that every partial derivative of every output of interest can be easily computed and validated. Even after significant development efforts, there may still be unresolved implementation issues related to nondifferentiability, incomplete convergence of iterative procedures, or magnitudes of variables small enough to be comparable to numerical error of the AD process. We find, however, that we can obtain derivatives for the variables of physical interest.

Currently, the differentiation of MATWS can be performed in the forward (or direct) mode, that is, by the chain rule following the flow of the code. The reverse (or adjoint) mode of differentiation, which constructs the chain rule by reversing the control flow, is also available. The significant differences between the two modes will be explained in more detail in Section 3. For now, we note that one run of the direct mode computation can produce partial derivatives of all outputs with respect to a single input. The reverse mode computation yields derivatives of a single output with respect to all inputs. Direct mode calculations are useful for factor importance analysis and construction of linear approximations of the model response. For applications with many control variables, the reverse mode AD is significantly more economical than both finite differences and forward-mode AD. For our existing work in uncertainty quantification of nuclear reactor simulations, computationally inexpensive adjoint differentiation may in fact be required. This general need for adjoint derivatives, based on our wider research interests, motivates the current stage of our project. However, our recent progress in performing differentiation in the adjoint mode is not discussed here because of the specialized and technical nature of the current work. We note only that the issues that need to be overcome have more to do with selection of optimal programming practices than with fundamental difficulties in applying automatic differentiation.

Because the subject of automatic differentiation is relatively new, the set of available complex examples is limited. One can intentionally write code that deceives AD tools, and the methodology for detecting such code is only in its development stages. Moreover, not all of the existing regularly developed code can be passed through AD tools: some data and control structures are unacceptable and have to be rewritten. doing so is particularly difficult with nuclear reactor simulation codes that may be written using older programming practices, only sparsely commented, and in many senses untouchable (i.e., modifications to the code must be minimal, and the mathematical structure cannot be modified because of the risk of compromising validity of the results). The question of how much code should be read and modified by a specialist implementing AD is largely open. In addition, adjoint differentiation is always harder to implement, and not all of the available AD packages have adjoint capability.

In view of the preceding remarks, we state that the primary goal of the work accomplished so far was to see whether AD can be performed at all. A current goal is to see whether AD can be performed at low computational cost. A major part of the development effort has gone into discovering and resolving implementation issues. For the next large example, we expect greater development efficiency, as well as a closer examination of how to achieve high numerical accuracy and how to restructure the code for optimal AD performance.

2 Motivation

Unlike a set of physical observations of the model, with sensitivities and parameter dependencies available only by statistical inspection, a simulation code is a sequence of analytic, explicit steps, open to guided or automatic examination. The analysis of performance of a numerical model can be done using a combination of statistical tools and examination of the algorithm.

Significant results with AD have been reported with fluid dynamics, aerodynamics, geophysical data assimilation and computational chemistry codes [2, 4, 3, 12, 17, 27]. To the best of our knowledge, this work is a first attempt to differentiate a nuclear reactor code with significant legacy components. The issues that arose when differentiating, and their solution, are of interest to both nuclear engineers and the computational scientists developing automatic differentiation software. We are actively collaborating with the OpenAD development group, our goal being to make the next releases of the tool applicable to a wider range of computational codes.

Our ongoing work is also a part of a larger effort to improve on the existing practices of uncertainty quantification, verification, and validation (i.e., estimation of likely variation in the numerical model outputs, comparison of the model performance with the design and experimental data) of numerical models of nuclear engineering, with applications to new code licensing and safety analysis. An important part of the model factor analysis is the efficient computation of model output derivatives with respect to inputs. Obvious uses of derivative information include estimating the model response to change in the inputs by linear approximation and ranking inputs by importance for model complexity reduction.

In addition, derivative information was used to improve the efficiency of newly developed approach to uncertainty quantification [23, 24]. We used the fact that model response can be estimated by a high-quality polynomial interpolation; that is, any calculation $y = F(x)$ where F is a complicated, possibly implicit procedure, can be approximated by an explicit formula $y \approx P(x)$, where P is a (high-order, multivariable) polynomial function on the (selected group of) inputs. The polynomial approximation is fitted by collocation based on a sample of inputs x_i and corresponding outputs $y_i = P(x_i)$. Each sample requires a full model run. If the gradient of the output is available at a fraction of the cost of computing the output, the approximation is also fitted to $\nabla_x y_i = \nabla_x P(x_i)$, effectively allowing the use of much fewer sample points and achieving higher quality of approximation. In our previous work, we demonstrated that for a simplified, generic model of the nuclear reactor core, this approach to uncertainty quantification is more precise than linear approximation, as well as computationally less expensive and providing greater analysis capacity than large-scale Monte Carlo sampling; more details are available in [23]. Applying the approach to a more complex simulation code is a logical next step.

The long-term goal is to be able to differentiate simulation codes of complexity close to SAS/SASSYS [6, 22]. As far as we know, it is possible in principle but requires substantial development efforts. We hope to develop most of the required methodology using the example of MATWS.

3 Automatic differentiation

Automatic (algorithmic) differentiation (AD) [14] is a technique to evaluate derivatives of functions defined by computer programs. The internal workings of AD are based on the fundamental observation that any function implemented in a computed program, regardless of its complexity, can be viewed as a finite sequence of elementary operations involving one or two arguments. Examples of such operations are assignment, addition, division, and other nonlinear elemental functions such as sin or exp, whose derivatives are known.

The first step in using AD is to identify the interesting inputs and outputs for the program, namely, the control and objective variables. For example, we chose for MATWS to compute the partial derivatives of the coolant, structure, and cladding temperatures (the objectives) with respect to various reactivity coefficients (inputs or control variables). In the text of the code, the chosen outputs of interest are

$$TCOOL, TCLAD, TSTRU, TFUEL,$$

representing coolant, cladding, structure, and fuel temperature, respectively. The chosen inputs (corresponding to physically significant sources of uncertainty) are

$$ALFARD, ALFACR, ALFAEX, ALFADP,$$

standing for radial core expansion, control rod driveline expansion, fuel axial expansion, and Doppler reactivity feedback coefficients, respectively.

Once the controls and objectives have been identified, the AD engine parses the initial program into a sequence of elementary operations and then computes the desired derivatives by using the chain rule of differential calculus.

Suppose that the original program $\mathcal{P}$ (implementing the *forward* model) computes

$$y = F(x) \, , \tag{1}$$

where $y \in \mathbb{R}^m$ are the objectives, $x \in \mathbb{R}^n$ represent the control variables, and $F : U \subset \mathbb{R}^n \to V \subset \mathbb{R}^m$ is a differentiable function, at least on the domain of interest (the smoothness assumption).

A kwy concept in AD is that of an *active* variable. Any variable defined inside $\mathcal{P}$ that depends on some component of x and influences some component of y is called an active variable. Conversely, any variables that do not meet this criterion are called *inactive*. No derivatives are computed for inactive variables.

AD can be implemented in two ways. One is through operator overloading, which uses specific language features that allow the usual operators such as $+, /, -$ to be redefined for certain variable types. For every active variable x defined in $\mathcal{P}$, a new variable of active type is defined that contains both the value of x and its derivative. Both the original variable and its derivative are propagated simultaneously with the help of the overloaded operators. The main advantage of AD through operator overloading is that the program remains almost unchanged, since all the derivative enhancements are done at compilation time. The only changes required in most cases are redefining the active variables to be of the composite type and initializing the right derivative values for the desired inputs. However, since Fortran 77 does not allow operators to be overloaded, we will not discuss this approach here. Important drawbacks of this approach to AD are the computational overhead of operator overloading, which increases the runtime of the program, and the fact that the reverse mode of AD (see below) is cumbersome to implement using this approach.

Another way of implementing AD is as a source-to-source transformation process. Given $\mathcal{P}$, AD tools such as ADIFOR [5], TAMC/TAF [9, 8, 10], OpenAD [28], and TAPENADE [18] yield an enhanced program $\mathcal{P}'$ that computes the original values as well as their derivatives, using the basic chain rule from differential calculus. Thus, the new program is enhanced with suitable data structures and operations to calculate and store the derivatives of the objectives with respect to the control variables.

Automatic differentiation has two basic *modes*, which refer to the directions in which the partial derivatives are propagated in the differentiated program.

Let us first examine how AD propagates derivatives in the *forward mode*. If we view the original program $\mathcal{P}$ as a sequence of k elementary statements (this is the viewpoint of the AD engine itself), then we can write (1) as

$$y = F_k(F_{k-1}(F_{k-2}(\ldots F_1(x)\ldots))) \, , \tag{2}$$

where $F_1, \ldots F_k$ are elementary vector functions (their components are $+$, sin, exp, etc.), whose derivatives are known. Clearly

$$F := F_k \circ F_{k-1} \circ \ldots \circ F_1(x) \, .$$

Then, upon invocation of forward mode AD, we obtain $\mathcal{P}'$, which computes, together with (2), the directional derivative

$$\dot{y} = \frac{\partial F_k}{\partial F_{k-1}} \cdot \frac{\partial F_{k-1}}{\partial F_{k-2}} \cdot \ldots \cdot \frac{\partial F_1}{\partial x} \cdot \dot{x} \tag{3}$$

for an arbitrary *seed* vector $\dot{x} \in \mathbb{R}^n$.

The *reverse mode* (also known in the literature as the *adjoint mode*) of AD reverses the control flow of the original program $\mathcal{P}$. Hence, $\mathcal{P}'$ computes the directional derivative

$$\bar{x}^T = \bar{y}^T \left(\frac{\partial F_1}{\partial x}\right)^T \cdot \ldots \cdot \left(\frac{\partial F_{k-1}}{\partial F_{k-2}}\right)^T \cdot \left(\frac{\partial F_k}{\partial F_{k-1}}\right)^T \tag{4}$$

for a user-chosen adjoint seed vector $\bar{y} \in \mathbb{R}^m$. We note that the elements of the Jacobians of the local transformations are called *elementary partial derivatives*.

For the sake of concreteness, let us consider a small example and highlight the differences between the two modes of (source-to-source) AD. Suppose $x \in \mathbb{R}^2$, $y = F(x) = F_2 \circ F_1(x) = \exp(x_1 \cdot x_2)$ and $\mathcal{P}$ reads

$$\begin{aligned} z &= x_1 \cdot x_2 \\ y &= \exp(z)\,. \end{aligned} \tag{5}$$

Then the forward mode yields $\mathcal{P}'$:

$$\begin{aligned} \dot{z} &= \frac{\partial z}{\partial x_1}\,\dot{x}_1 + \frac{\partial z}{\partial x_2}\,\dot{x}_2 = x_2\dot{x}_1 + x_1\dot{x}_2 \\ z &= x_1 \cdot x_2 \\ \dot{y} &= \frac{\partial y}{\partial z}\,\dot{z} = \exp(z) \cdot \dot{z} \\ y &= \exp(z)\,. \end{aligned} \tag{6}$$

Similarly, the program produced by adjoining $\mathcal{P}$ looks as follows:

$$\begin{aligned} z &= x_1 \cdot x_2 \\ y &= \exp(z) \\ \bar{z} &\mathrel{+}= \exp(z) \cdot \bar{y} \\ \bar{x}_1 &\mathrel{+}= x_2 \cdot \bar{z} \\ \bar{x}_2 &\mathrel{+}= x_1 \cdot \bar{z}\,. \end{aligned} \tag{7}$$

It is now apparent that

$$\dot{y} = \frac{\partial F}{\partial x}\,\dot{x} \tag{8}$$

and

$$\bar{x}^T = [\bar{x}_1 \;\; \bar{x}_2] = \bar{y}\,\frac{\partial F}{\partial x}\,. \tag{9}$$

3.1 Complexity of forward vs. reverse mode

A simple dimensional analysis of (3) vs. (4) reveals that forward-mode differentiation is preferable when $m \gg n$, while the reverse mode is the better choice when the number of control variables is much larger than that of the objectives (i.e., $n \gg m$). The latter case is found, for example, in aerodynamic optimization [11] or in ocean modeling [19, 7], where one may consider thousands to millions of control variables, but only a small number of objective functionals. In the case where $m \approx n$, the forward mode usually proves more economical, since it avoids issues such as checkpointing [16] or forward trajectory recomputations.

Note that, for nonlinear models (i.e., nonconstant local Jacobians), the adjoint mode requires either the storage or the recomputation of several forward model variables. The reason is that the control flow of $\mathcal{P}$ is reversed upon adjoint differentiation. For nonlinear F, the evaluation of the elementary partial derivatives in $\partial F_{j+1}/\partial F_j$ requires the values of intermediate variables computed by F_j (inside $\mathcal{P}$). Hence, the values of these variables must be either taped and restored (this checkpointing [14] comes at the expense of an increased memory/disk footprint for $\mathcal{P}'$) or recomputed on the fly when needed (leading to an increased runtime for $\mathcal{P}'$). By default, OpenAD stores intermediate values inside memory buffers (also called tapes). The default behavior of TAMC is to recompute all forward variables and use them as needed, but this can be customized by introducing tape directives [8].

Assuming the computational cost of a forward simulation is $\mathcal{C}_F$, Griewank [14] gives an encouraging upper bound for the cost of one adjoint simulation:

$$\mathcal{C}_A \leq 5 \times \mathcal{C}_F \, . \tag{10}$$

Further reductions are usually achieved in practice by judicious checkpointing and compiler optimizations, which can lower the cost of computing a function gradient ($m = 1$) to around $2 \times \mathcal{C}_F$. The advantage over the finite-difference method, whose cost is roughly $(n+1) \times \mathcal{C}_F$, is evident.

3.2 Vector mode

There is also a way to efficiently compute submatrices of the Jacobian $\partial F/\partial x$. Using the notation in (1), one can describe the behavior of the vectorized AD modes as follows. Given a seed matrix $\dot{X} \in \mathbb{R}^{n\times k}$, the *vector forward mode* yields

$$\dot{Y} = \frac{\partial F}{\partial x}\dot{X} \, , \tag{11}$$

while the *vector reverse mode* calculates

$$\bar{X}^T = \bar{Y}^T \frac{\partial F}{\partial x} \, , \tag{12}$$

from a user-chosen seed matrix $\bar{Y} \in \mathbb{R}^{m\times k}$. Using vectorized AD can substantially reduce the overall time needed to calculate the k directional derivatives, compared with calculating the derivatives obtained by k runs of (3). The reason is that all forward calculations and intermediate variables (such as the arguments of the local Jacobians) are shared and used for all derivative directions at once.

3.3 Higher derivatives

Apart from Jacobians or first order directional derivatives, one may wish to compute higher-order derivatives such as Hessians or Taylor series expansions. This is achievable through multiple invocations of AD on $\mathcal{P}$. For example, a Hessian-vector product can be computed through a forward-over-reverse differentiation. From these second-order directional derivatives, one can reconstruct the whole Hessian tensor, if required. However, the cost of such a calculation can grow rapidly with both m and n [14].

3.4 Advantages of Automatic Differentiation

Automatic Differentiation has several advantages over other numerical or symbolic differentiation techniques. The differentiated code is generated automatically, with minimal user intervention prior to the actual differentiation. AD is also flexible, in that it allows the user to choose any subset of inputs and outputs from the existing set in $\mathcal{P}$. The derivatives computed by AD can usually be made as accurate as the quantities computed by the original program. Special care is needed with derivatives of iterative solvers [1, 15], however, which usually converge at a slightly slower rate than do the original iterations.

Another important advantage of AD over finite differences is that the AD derivatives do not suffer from truncation errors or catastrophic cancellation. Moreover, unlike symbolic differentiation, AD does not lead to expression blow-up [13].

Explicit and implicit Runge-Kutta integration methods of arbitrary order are known to be convergent under automatic differentiation [25]. Moreover, their differentiated versions retain the order of accuracy of their original counterparts. Linear multistep methods (LMMs) do not preserve consistency under adjoining, in general. However, the discrete adjoints of LMMs do converge to the adjoint ODE solution at the initial time, and with the same order of accuracy as the original linear multistep method [26].

4 Code preprocessing

The MATWS source code consists of $\approx$ 10k lines of Fortran 77, The following programming features make the application of AD to such code difficult.

- CPU-dependent (therefore nonportable) sections in the code
- Old, deprecated, or non-standard language constructs
  - The Fortran `EQUIVALENCE` construct causes the automatic code analysis to be less precise in retaining conservative correctness; consequently, the derivative code becomes less efficient.
  - `COMMON` blocks (replaced in the newer Fortran standards by module variables) do not allow a default initialization of active type variables that is convenient and fast for the adjoint mode of automatic differentiation.
  - Subroutines that have variable number of parameters; in the newer Fortran standards this issue is addressed by optional dummy arguments and calls with name lists.

– Direct memory references, variable offset computations, and memory copy operations are difficult or impossible to address in the code analysis without leading to a significant loss of precision while attempting to retain conservative correctness; consequently, the derivative code will becomes less efficient.

ADIFOR requires standard Fortran 77 code, so one cannot have the aforementioned nonstandard constructs, such as common blocks with the same name but different lengths, or subroutine calls with variable number of arguments. Several modifications are needed before running ADIFOR:

- Remove CPU or OS dependent code sections, variable address computations, system calls, and the like. These do not play a role when computing derivatives of output quantities, and are used used for input and output formatting, for obtaining user names, user IDs, and so on.
- Define one subroutine for each call as needed (since subroutines with a variable number of arguments are not supported). There remains an issue with subroutines whose stack variables have persistent state (i.e., `SAVE` variables, which include all variables that are declared with an initial value), because a duplication of the subroutines means that there is no consistent persistent state between the subroutine duplicates. To bypass this problem, one could move all those variables into a special common block that is shared between the subroutine duplicates in question, but their values then would need to be initialized by some separate routine to be called at the beginning of the program.
- Rename common blocks with inconsistent sizes between subroutines. We also enforced strong typing in nonactive subroutine calls (e.g. `call foo(x)`) through explicit casts to a temporary variable and copying back of the value where needed and appropriate, i.e. `t=DBLE(x); call foo(t); x=INT(t)`. Because type mismatches are inherently unsafe and this modification is trying to add some type safety, one has to first ascertain the intention of the call to yield the desired effect in a type safe fashion.

4.1 Fortran EQUIVALENCE statements

An `EQUIVALENCE` statement is used to specify several variables that share the same memory storage. It is a relic from the epoch before virtual memory management (some sort of memory management technique was needed to save memory space). The AD tools support `EQUIVALENCE`s at varying degree typically by generating the corresponding `EQUIVALENCE` statements for the tangent derivatives. Thus, those derivatives will share the same memory storage, mirroring the relationship between the forward model variables.

MATWS uses `EQUIVALENCE` statements for two purposes:

- I/O procedures. The `EQUIVALENCE`s are used to define variables that point to the beginning of a `COMMON` block. Then, using the intrinsic `LOC` function, the program computes the length of the common block in bytes. These length variables (`NOINTC` and `NOFLTC`) are used when reading the input file chunk by chunk in the temporary buffers. The AD engine is oblivious to this read-in procedure.
- Initializing constant-element arrays. Several constant element arrays of the same size share the same memory space (thus, their contents are identical). If the variables in question are

active, then one should replace the EQUIVALENCEs by simple array allocations and element-wise assignments (or vector assignments in Fortran 90).

4.2 Direct-memory copy operations

The AD engine cannot differentiate arithmetic operations (such as assignments or addition and subtraction) which are performed through direct memory access (e.g., using something like the nonstandard intrinsic LOC function to get the beginning address and offset of a given array). One needs to rewrite these code sections in terms of explicit array operations, if they involve active variables.

4.3 Fortran COMMON blocks

All the aforementioned AD tools support common blocks. Suitable common blocks will be defined for the tangent linear and adjoint variables upon differentiation. For example, if $\mathcal{P}$ contains

```
COMMON /ABLOCK/ A
DOUBLE PRECISION A(2)
```

and A is an active variable, then $\mathcal{P}'$ will contain something similar to the following.

```
COMMON /DABLOCK/ dA
DOUBLE PRECISION dA(2)
```

or with OpenAD

```
COMMON /DABLOCK/ dA
type (active) :: dA(2)
```

4.4 Implicit typing and statement functions

All tools support the IMPLICIT statement. TAMC/OpenAD will define all variables explicitly in the differentiated code $\mathcal{P}'$. ADIFOR will mirror the definitions in the original program $\mathcal{P}$.

OpenAD does not yet support statement function definitions. Hence, such definitions must be recoded into explicit function or subroutine definition before invoking the OpenAD script. For example, MATWS contains the following statement function definition inside subroutine INIT.

```
DOUBLE PRECISION TSL(8)
DOUBLE PRECISION P, H
     ...
TSCFUN(P,H) = (TSL(1)+H*(TSL(2)+H*(TSL(3)+H*TSL(4))) + P*(TSL(5) +
      H*(TSL(6)+H*(TSL(7)+H*TSL(8))))-32.D0)*(5.D0/9.D0) + 273.D0
     ...
TGUESS=TSCFUN(PWAVEP,HGUESS)
```

Before OpenAD is run on this code, one must transform this implicit definition into a subroutine definition.

```
SUBROUTINE TSCFUN(P,H,RES_TSCFUN)
DOUBLE PRECISION P
DOUBLE PRECISION H
DOUBLE PRECISION RES_TSCFUN
...
RES_TSCFUN = (TSL(1)+H*(TSL(2)+H*(TSL(3)+H*TSL(4))) + P*(TSL(5)+
   H*(TSL(6)+H*(TSL(7)+H*TSL(8))))-32.D0)*(5.D0/9.D0) + 273.D0
END SUBROUTINE TSCFUN
...
```

Note that statement functions are deprecated in the newer Fortran standards.

4.5 Validation of the modified code

We acknowledge the concern that the changes performed on the code before the AD tools become applicable may put the validity of the simulation in question. In many numerical simulations one can observe numerical "jitter" caused by changing the hardware platform, the compiler, or compiler optimization level without actually changing the source code. It is necessary to verify that the code that has been modified according to the above description produces output that is within an acceptable tolerance from the original. Some of the necessary modifications address programming techniques that are considered inherently unsafe, thus increasing the portability and maintainability of the code.

The Fortran 90 version of MATWS has been validated against the original Fortran 77 code by comparing outputs corresponding to the nominal input values. Outputs from both programs were found to be identical. We have not yet validated output consistency for the perturbed inputs that are used in the finite-difference approximations. However, we expect those to be the same for both code versions. We also note that ADIFOR was able to differentiate the original Fortran 77 code after some minor modifications done to `inactive` functions. Since the ADIFOR derivatives in Table 1 agree remarkably well with both finite differences and the rest of the AD tools, we are confident that the results of the differentiation are indeed correct as given.

5 Code postprocessing

We have not encountered implementation issues that definitely require code postprocessing. We find it helpful, however, to develop methodology at an abstract level.

If the model is successfully passed through the automatic differentiation tools but the derivative values and a finite-differences check do not converge, the augmented code can be analyzed for nondifferentiable routines. Where necessary one may modify the source code to smoothen or otherwise correct the nondifferentiability before reapplying AD. The AD-generated derivative values are correct up to machine precision and at nondifferentiable points still can produce an element of the generalized gradient. We note, however, that certain programming constructs have unintended side effects on the derivative computation. Among these are branching, table lookups, and conversion between floating-point values and integers. The AD community has accumulated some experience regarding these constructs and how to detect and treat them.

We formulated a few rules for manual smoothing. If an analytic expression is not smooth, a smooth approximation should be constructed and differentiated (geometrically, the required operation is the rounding of corners in function plots). If there is no analytic expression and the function evaluation consists of a lookup table, a smooth interpolation should be constructed and differentiated (a stepwise function, made smooth by some rounding at the corners, is probably better than a least-squares-fitted polynomial curve). If everything else fails and the procedure does not have an adequate analytic differentiable approximation, a finite-differences approximation should be calculated for this elementary derivative.

Correspondingly, our ideas on resolving incorrect derivative information obtained by AD differentiation of the complete code are as follows:

- Compute partial derivatives for some code sections through finite differences, or provide some smooth (and valid) model approximation. This procedure is easily applicable to the forward mode (but not quite so for the adjoint program). It involves low computational overhead, and is simple to implement once the troublesome derivative code has been identified (e.g., can isolate that code in a subroutine). The steps are as follows:
    1. Differentiate the code in its original form.
    2. Identify the sections of code that are non-smooth or contain spurious AD derivatives.
    3. Isolate that piece of code in subroutines if possible (just for convenience).
    4. Replace the code with a smooth approximation of the original model, or compute the necessary partial derivatives through finite differences.
- Compute the derivative, not at the current point in parameter space, but at a number of nearby points. Build a smooth interpolation.
- If a derivative iteration does not converge, increase the number of iterations and/or modify the convergence criterion to explicitly take the derivatives into account. This process involves low development cost.
- If there is a control structure switching between algebraic expressions, either ignore it, or build a smooth interpolation based on points away from the switch.
- For derivative iterations, increase the number of iterations, and/or modify the convergence criterion to take into account the derivatives.

6 Checking for derivative correctness

We verify the automatic differentiation results using finite differences. Recall that $\mathcal{P}$ computes $y = F(x)$. Then, for a suitably chosen step size h and any direction $\dot{x}$ (assuming F is sufficiently smooth at x), the following Taylor approximation holds:

$$F(x + h\,\dot{x}) - F(x - h\,\dot{x}) \approx 2h \cdot \left(\frac{\partial F}{\partial x}(x)\,\dot{x} \right) \ . \tag{13}$$

Table 1: Comparison of different estimates for the derivatives

Software	$\frac{dTCOOL}{dALFARD}$	$\frac{dTSTRU}{dALFARD}$	$\frac{dTCLAD}{dALFARD}$	$\frac{dTFUEL}{dALFARD}$
ADIFOR	17468.4511373	17468.3752648	17639.4606974	18312.5474227
OpenAD	17468.4511372	17468.3752647	17639.4606974	18312.5474227
TAMC	17468.4511392	17468.3752667	17639.4606995	18312.5474248
Complex approximation	17468.4511372	17468.3752647	17639.4606974	18312.5474226
Finite differences	17468.4315994	17468.3557532	17639.4409493	18312.5269537

The directional derivative in the right-hand side of this expression is computed through AD; the left-side uses a central difference scheme requiring two model runs (once the optimal perturbation h has been found).

The derivatives can also be computed by using a complex-step derivative approximation procedure. This approach is based on a simple result from complex analysis:

$$\frac{\partial F}{\partial x} \approx \frac{Im\left[F(x + \imath h)\right]}{h} \; . \tag{14}$$

Since there is no possibility of catastrophic cancellation, one can take h smaller than machine epsilon and get accurate derivative approximations. In Table 1, we show a typical comparison of derivative values, for a single input and a few outputs of interest. In general, for the previously selected inputs and outputs of interest, the numerical tests consistently show agreement (with the relative error of 0.01% or better). Note that, before attempting (14), some preprocessing of the forward simulation code is required (see [20] for guidance and tools).

We note that the validation process is not complete. We were able to locate one output variable ($AMPI$ in the code) for which the finite-differences derivative disagrees with the AD estimation (by a factor of exactly 100, which may indicate scaling in the input, or in the output, that our finite differences procedure does not process correctly). We hope to resolve all such issues through the use of tracing tools; familiarity with the meaning of each variable may also help.

7 Conclusions

We conclude this report with a perspective on the future of the project and the related efforts. Much of our understanding is based on recent discussions with the AD development community.

In an ideal situation, the tools of automatic differentiation should be applied only to structured codes. There are strong suggestions that any new code should be developed with automatic differentiation in mind, perhaps even simultaneously with its AD-augmented version. If that were to become a standard, effectively all new software would be capable of outputting its own sensitivity information, with minimal additional effort. In our field of study, however, the codes of interest are likely to have old, unstructured, components sometimes without comprehensive documentation and without general permissions to modify the code. The only allowed invasion into the code's structure is at the postprocessing stage, where we can replace the functions automatically developed to output derivatives by smooth, or more precise forms.

Nevertheless, the significance of the widely known issues with applying automatic differentiation to codes with significant legacy base may be exaggerated. Such commonly identified difficulties with

Fortran program structures (flow control operators, equivalences, common blocks) are handled nicely by the existing tools. Once we demonstrate that our complicated example may also be differentiated in the adjoint mode, our next goal is to apply tracing tools to identify nondifferentiable segments of the codes and to develop widely applicable methodology for constructing and debugging AD-augmented versions of (almost) any code. In the long term, we would also like to achieve complete, automatic validation of AD results.

Acknowledgments

We would like to thank Andrew Lyons and the rest of the AD team in Mathematics and Computer Science Division, Argonne, for their continuous help with OpenAD. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] T. Beck, *Automatic differentiation of iterative processes*, in ICCAM'92: Proceedings of the Fifth International Conference on Computational and Applied Mathematics, Amsterdam, The Netherlands, 1994, Elsevier Science Publishers B.V., pp. 109–118.

[2] C. Bischof, C. Corliss, L. L. Green, A. Griewank, K. J. Haigler, and P. A. Newman, *Automatic differentiation of advanced cfd codes for multidisciplinary design*, Journal of Computing Systems in Engineering, 3 (no. 6) (1992), pp. 625 – 638.

[3] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and E. Slusanschi, *Efficient and accurate derivatives for a software process chain in airfoil shape optimization*, Future Gener. Comput. Syst., 21 (2005), pp. 1333–1344.

[4] C. H. Bischof, H. M. Bücker, and A. Rasch, *Sensitivity analysis of turbulence models using automatic differentiation*, SIAM J. Sci. Comput., 26 (2005), pp. 510–522.

[5] C. H. Bischof, A. Carle, P. M. Khademi, and A. Mauer, *The ADIFOR 2.0 system for the automatic differentiation of Fortran 77 programs*, Tech. Report Preprint ANL/MCS–P481–1194, Argonne National Laboratory, 1994.

[6] F. Dunn and F. Prohammer, *SASSYS LMFFBR systems analysis code*, Math. Comp. Simul., 26 (1984), pp. 23–26.

[7] F. Fang, M. Piggott, C. Pain, G. Gorman, and A. Goddard, *An adaptive mesh adjoint data assimilation method*, Ocean Modelling, 15 (2006), pp. 39–55.

[8] R. Giering, *Tangent Linear and Adjoint Model Compiler, Users Manual 1.4*, 1999.

[9] R. Giering and T. Kaminski, *Recipes for Adjoint Code Construction*, ACM Trans. Math. Software, 24 (1998), pp. 437–474.

[10] ——, *Applying taf to generate efficient derivative code of Fortran 77-95 programs*, Proceedings in Applied Mathematics and Mechanics, 2 (2003), pp. 54–57.

[11] R. Giering, T. Kaminski, and T. Slawig, *Generating efficient derivative code with taf adjoint and tangent linear euler flow around an airfoil*, Future Gener. Comput. Syst., 21 (2005), pp. 1345–1355.

[12] L. L. Green, P. A. Newman, and K. J. Haigler, *Sensitivity derivatives for advanced cfd algorithm and viscous modeling parameters via automatic differentiation*, J. Comput. Phys., 125 (1996), pp. 313–324.

[13] A. Griewank, *On automatic differentiation*, Tech. Report CRPC-TR89003, Center for Research on Parallel Computation, Rice University, 1989.

[14] ——, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2000.

[15] A. Griewank, C. H. Bischof, G. F. Corliss, A. Carle, and K. Williamson, *Derivative convergence for iterative equation solvers*, Optimization Methods and Software, 2 (1993), pp. 321–355.

[16] A. Griewank and A. Walther, *Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation*, ACM Trans. on Math. Software, 26 (2000), pp. 19–45.

[17] L. Hascoët and B. Dauvergne, *Adjoints of large simulation codes through automatic differentiation*, REMN Revue Européenne de Mécanique Numérique / European Journal of Computational Mechanics, 17 (2008), pp. 63–86.

[18] L. Hascöet and V. Pascual, *TAPENADE 2.1 User's guide*, Tech. Report 0300, INRIA, Sophia Antipolis, France, 2004.

[19] P. Heimbach, C. Hill, and R. Giering, *An efficient exact adjoint of the parallel mit general circulation model, generated via automatic differentiation*, Future Gener. Comput. Syst., 21 (2005), pp. 1356–1371.

[20] J. R. R. A. Martins, P. Sturdza, and J. J. Alonso, *The complex-step derivative approximation*, ACM Trans. on Math. Software, 29 (2003), pp. 245–262.

[21] E. E. Morris, *Uncertainty in unprotected loss-of-heat-sink, loss-of-flow, and transient-overpower accidents*, Technical Report ANL-AFCI-205, Argonne National Laboratory, 2007.

[22] G. Palmiotti, J. Cahalan, P. Pfeiffer, T. Sofu, T. Taiwo, T. Wei, A. Yacout, W. Yang, A. Siegel, Z. Insepov, M. Anitescu, et al., *Requirements for advanced simulation of nuclear reactor and chemical separation plants*, Tech. Report ANL-AFCI-168, Argonne National Laboratory, 2006.

[23] O. Roderick, M. Anitescu, and P. Fischer, *Polynomial regression approaches using derivative information for uncertainty quantification*, Nuclear Science and Engineering, (2010). To appear.

[24] O. Roderick, M. Anitescu, P. Fischer, and W.-S. Yang, *Stochastic finite-element approach in nuclear reactor uncertainty quantification*, Transactions of American Nuclear Society, 100 (2009), pp. 317–318.

[25] A. Sandu, *On the properties of runge-kutta discrete adjoints*, in International Conference on Computational Science (4), 2006, pp. 550–557.

[26] ——, *On consistency properties of discrete adjoint linear multistep methods*, Tech. Report TR-07-40, Virginia Polytechnic Institute and State University, 2007.

[27] R. Steiger, C. H. Bischof, B. Lang, and W. Thiel, *Using automatic differentiation to compute derivatives for a quantum-chemical computer program*, Future Gener. Comput. Syst., 21 (2005), pp. 1324–1332.

[28] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, and C. Wunsch, *OpenAD/F: A modular open-source tool for automatic differentiation of fortran codes*, ACM Trans. on Math. Software, 34 (2008), pp. 1–36.