ANL/MCS-TM-328

Derivative-Based Uncertainty Quantification: Automatic Differentiation Tools for SAS

Mathematics and Computer Science Division

ANL/MCS-TM-328

Derivative-Based Uncertainty Quantification: Automatic Differentiation Tools for SAS

prepared by
O. Roderick, M. Anitescu, and Jean Utke
Mathematics and Computer Science Division, Argonne National Laboratory

September 30, 2012

ABSTRACT

Automatic differentiation, or more properly algorithmic differentiation (AD), is a technique for efficiently computing accurate derivatives for numerical models. It is based on augmenting the code with partial derivatives of elementary mathematical operations on important variables and retracing the calculation flow (in forward or reverse direction) to assemble derivatives by chain rule. The relative computational overhead associated with AD is bounded, independent of dimension, and is largely independent of the mathematical model.

In our larger body of work on advanced uncertainty analysis of simulation models of nuclear engineering, AD serves as the driving element behind such methods as polynomial regression with derivatives, gradient-enhanced universal kriging, and sensitivity-based dimensionality reduction of the uncertainty space. In fact, the main alternative to using AD to get sensitivity information is hand coding of direct and adjoint derivatives, which is always a significant development effort and often cannot be expected from the developers.

In this report, we discuss the findings and intermediate benefits of the latest effort to enable algorithmic differentiation of the SHARP safety code SAS. This effort was planned as an exercise to demonstrate the effectiveness of AD on simulations of professional interest. The subject is a legacy code comprising 120,000 lines of uncommented Fortran 77 code; it thus presents significant challenges for manual analysis. An intermediate outcome of the preparation work required for AD is a semi-automatically generated code annotation, with identification and characterization of code features in the context of AD.

While no principal, mathematical model-related reason prevents algorithmic differentiation for SAS, certain features in model implementation pose steep technical hurdles. We report the categories of the problematic features found in SAS and recommend possible remedies, amounting to a future effort in code rewriting.

Because the full capability for differentiation of SAS was not achieved, we redirected the effort to differentiation of the SHARP component neutronics code UNIC. We report on the outcome—differentiation in forward mode achieved—and outline other capabilities that can now be acquired and verified.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The modern field of nuclear engineering relies on complex numerical simulation models for reactor prototype development, licensing, safety analysis, and performance optimization. Automatic differentiation, or more properly algorithmic differentiation (AD), of simulation codes is arguably the most developed method of intrusive analysis [1,2]. Enabled automatic differentiation provides an immediate benefit of computationally inexpensive sensitivity analysis of the code. In our past and ongoing work we have also shown that differentiation capability can be used to significantly accelerate and improve the response surface methods for uncertainty quantification, as applied to nuclear engineering simulations. AD was used as the driving element behind such methods as polynomial regression with derivatives [3], gradient-enhanced universal kriging [4], and sensitivity-based dimensionality reduction of the uncertainty space [5]; it also fits well with the current work on multifidelity uncertainty analysis methods.

This report outlines the necessary building blocks for implementing AD in the systems code SAS [6]. We then explain which programming idioms in SAS pose problems for applying AD tools. We provide guidelines for SAS code changes to remove these obstacles and present two approaches for a remedy, along with a brief description of how regression and validation tests would accompany the proposed remedies.

This work was completed as part of the reactor performance and safety code development project, in fulfillment of milestone M4MS-12AN0603242, funded under the Nuclear Energy Advanced Modeling and Simulation (NEAMS) program of the U.S. Department of Energy Office of Nuclear Energy.

We emphasize that problems with the SAS code discussed in the report originate from past design decisions made to accommodate various software and hardware restrictions at the time. In particular, they do not reflect current SAS development work, which is no longer bound by the same external restrictions but remains, to some extent, confined by the past decisions. Redesigning portions of SAS code has long been desired by developers, in order to improve th portability, scalability, and effectiveness of maintenance; permit new features; and enable complier optimization. We argue that a partial redesign with specific criteria for AD will yield the same desired improvements and is necessary to enable AD capability.

We also stress that preparation work involved in algorithmic differentiation of SAS yielded an automatically generated report of code features that have relevance outside the context of AD. We note that manual annotation and feature identification for approximately 120,000 lines of uncommented code would be an impractical task. Even without full capability for differentiation, we suggest that automatic analysis provided by AD tools therefore should be used by code developers within NEAMS.

Many of the restrictions referred to in the report do not apply to the SHARP code UNIC [7], which is organized differently and in many ways is more modern and compliant with coding standards. We were therefore able to demonstrate capability for differentiation in a much shorter time. Current work can be used as a basis for advanced uncertainty analysis and stochastic optimization for codes such as UNIC, SAS, and possibly other SHARP components. Enabling differentiation capabilities fits well into a larger body of work on advanced verification, validation, and uncertainty quantification [8,9].

2 Method Description

We explain in this section the motivation for using AD and then describe the basic components of AD. We identify the challenges raised by former programming practices and describe how AD can be used effectively in the UNIC code.

2.1 Motivation for automatic differentiation of simulation models

For a realistic, high-resolution simulation model of a physical system, we can expect the dimension of uncertainty to be large, making differentiation by finite-differences schemes unpractical. At the same time, the mathematical model behind the simulation will not be explicitly shown in the code; even with access to underlying equations and notes on designer decisions, hand coding of the derivative (or, construction of the adjoint code) can be a daunting task. For practical purposes, the only real options for getting at least partial derivative information (with respect to at least some inputs of interest) are implementing automatic differentiation or working with the code design team from the beginning of development to output sensitivities as a part of the code's main functionality.

The theoretical motivation for AD is a computability theory result by Griewank [10], stating that there does not exist a sequence of elementary functions for which the additional computational effort required to evaluate the gradient will be above a certain limit. The estimated limit of 500% (relative to run time of the unmodified code) is cheaper than finite differences at a dimension of parameter space equal to 3 or above. At high dimensions currently being considered for applied tasks on simulation models, this overhead is relatively negligible.

The value of derivative information for sensitivity analysis and error analysis on *deterministic* ODEs and PDEs is clear. Many techniques that would otherwise require additional cumbersome constructions or estimates are essentially development effort free if the gradient is available (e.g., gradient-based optimization searches, perturbation-theory based estimates, intelligent adaptive sampling of parameter space).

Our previous work on uncertainty analysis (for deterministic systems of differential-algebraic equations with *stochastic* parameters) has shown additional value in obtaining gradients. Specifically, we have shown that components of the gradient can be used as additional fitting conditions in multivariate regression techniques and stochastic processes-based machine learning; the result was that surrogate response to uncertainty, which would normally require many code evaluations, could be constructed by using outputs and gradients at <10 points in the parameter space. In Figure 1, we provide an illustration from recently published research on gradient-enhanced kriging techniques: a confidence interval for order statistics of a complex simulation model (SAS subset MATWS) is correctly placed using a total of 8 model runs [11,4]. Normally, this task would require hundreds of code evaluations.

Another important reason for AD analysis that will become clear in the following text is that preparation work required to pass the code through AD doubles as an extensive code analysis that arguably could not be reproduced by manual code review and annotation. This has implications for a wider range of tasks of verification, validation, ensuring standards compliance, and maintainability and portability of the code.

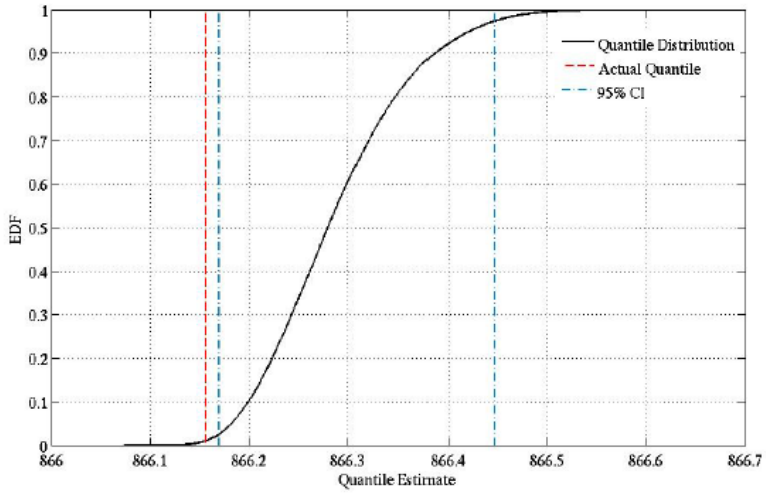

Figure 1: 95% confidence interval for 95th percentile, training set size 8

2.2 Automatic differentiation building blocks

Algorithmic differentiation starts by analyzing data dependencies from designated input variables to designated output variable. Next, it augments the program data with new data to hold the derivative values to be computed. Then, it creates new statements to logic that perform the computation by considering each elemental numerical operation applied to the program data that lies on some dependency path from the input to the output. Regardless of the AD tool one might apply to the SAS source code, AD comprises two fundamental building blocks.

Data dependency analysis and data augmentation. The computation of derivatives along with the original model computation incurs an overhead in memory and compute time. Therefore one often has to restrict derivative computation of certain outputs with respect to certain inputs. Dependency analysis filters out all the computation not on a path from the inputs to the outputs. The filtered-out computations are called passive computations, and the involved variables are passive variables. In contrast, all (floating-point) values computed on dependency paths from the designated inputs to the designated outputs are called active; that is, they are evaluated in active computations involving active variables. This code analysis can be precise when all the computed values in the program are held in distinct variables with a fixed semantic meaning (e.g., temperature or pressure). Whenever the data model implemented in the program merges the storage of semantically distinct values into a single entity (e.g., a large “work array” storing temperature and pressure in different index ranges), the currently implemented analyses cannot maintain the semantic separation. Conservatively correct assumptions have to be made; for example, all merged values are considered to be on the same dependency path that was established for one of the merged values. This process incurs overhead in storage and computation. Storage merging also occurs through the use of equivalence and (blank) common blocks.

For each active value (i.e., computed on a dependency path from the input to the output variables) the AD tool must make space to hold this value in memory. The computed value

must be held in some program floating-point variable, say, **r**. The AD tool can provide the space by changing the type of **r** to an active type that holds the original value as **r%v** and the corresponding derivative as **r%d**, or it can create a corresponding variable with a uniquely adorned name (e.g., **r_ad**) to hold the derivative value.

A semantically meaningful derivative computation can be generated only for arithmetic operations on floating-point values. When floating-point values are taken as bit patterns and reinterpreted to another type (as can be done through *transfer*, *equivalence*, F77-style parameter passing with deliberately mismatched types), there is no conservatively correct algorithm to generate derivative logic for any operations on the reinterpreted bit patterns. Even if no operations are performed, however, it is unclear whether and how the bit patterns of the corresponding derivative value should be stored and reinterpreted. The (frequently incorrect) fallback assumption would have to be that any conversion away from the floating-point representation signals that there is no differentiable dependency.

Data flow reversal. For a numerical operation $u = \phi(...,v,...)$ in the model in question, the basic adjoint propagation applied to each argument v can be written as $\bar{v} = \bar{v} + \frac{\partial \phi}{\partial v} u$. It is applied to all built-in numerical operations ϕ on active data. During the forward execution, the left-hand side u depends on the argument v. The dependency for the corresponding *adjoint* quantities is reversed in that $\bar{v}$ depends on $\bar{u}$. Applying this rule for the sequence of all statements in a program means that the data flow needs to be reversed for the whole program. Implementing the data flow reversal by source transformation implies control flow reversal and reversal of all statement sequences (including subroutine calls).

We note that the adjoint statement refers to the partial derivatives $\frac{\partial \phi}{\partial v}$. Given the overall reversal and the typical overwriting of variable values, one can see that computing the partials requires variable values in the forward order, while the partial values are required in the reverse order. To access the partial values, one may choose to store them on a stack, but doing so implies a stack size proportional to the number of floating-point operations executed at run time. To mitigate the stack size, one stores the partials only for sections and restarts computation from checkpoints. Optimized checkpointing schemes do exist, but for long computations these schemes require considerable space, and so checkpoint sizes have to be minimized and are taken as "application"-level checkpoints (as opposed to system-level checkpoints, with the entire process state). They are determined by the AD tool in terms of the model data. (Re)storing application-level checkpoints requires the data model to allow code generation to (de)serialize part of the model state. Minimizing the checkpoint size requires precise data dependency analysis for the same reason mentioned before.

2.3 Problematic programming practices

Most of the problematic Fortran idioms reported here are consequences of now-obsolete limitations in the language, compiler technology, or hardware; they are no longer needed, or a better alternative exists in terms of source code maintainability and expected runtime performance. We therefore attempted to reduce the source code base to be adjoint transformed by starting with model setups that do not exercise the full functionality of SAS. Through runtime profiling we found a reduction to a set of 308 distinct executed subroutines. Filtering

out ancillary logic for logging and initialization further reduced that number to 220. Given the existing implementation, collecting all the source files for the 220 routines and including the referenced modules results in a source code base of about 60K lines of code, or roughly 30% of the entire code base of SAS. Through preprocessing, the code size to be transformed grows to about 140K lines of code. The rationale for the reduced setup was the hope of implicitly excluding some of the code exhibiting the earlier identified problems. Unfortunately, that was not the case because the issues proved to be pervasive in SAS. The remainder of this section highlights the problems by category.

Data model. Most of the data is held in common blocks. A migration of the data to module variables and user-defined types has started but is still in its initial phase.

(P1) A serious problem is a repeated pattern using equivalence of the respective first element in a given common block to an array variable, as done in the following example:

COMMON /block/ e1, e2, e3

! etc.

REAL(KIND=KIND(1.0d0) :: array(1)

EQUIVALENCE (e1,array)

Subsequently, the *array,* which is declared to be of length 1, is used to access the data in the entire *block* by using indices >1 up to some implicitly agreed-upon upper bound representing the "size" of the block. If one assumes that all the elements in *block* are declared as double-precision floating-point numbers, an alignment restriction now requires all elements of *block* together with *array* to be either passive or active (denote this **P1a**). Consequently, there exists no consistent way to reduce the overhead for selections of subsets of inputs and outputs. The use of indices for *array* that are outside its declared bounds violates the Fortran standard. There is, however, no reliable compile time check to enforce the standard's restriction, and therefore such usage patterns have been common practice. We note that this practice prevents most runtime array bounds checking commonly implemented as a sanity check and debugging helper option in many compilers (**P1b**). At the same time, because there is neither a syntactic hint at the alignment requirement nor a compile time check, an AD transformation has to assume that any equivalencing implies such alignment restrictions even if, in reality, there are none. Many, but not all, common blocks in SASSYS have elements of a single type. If *block* contains elements of different types and active data, then a misalignment is virtually assured (**P1c**).

(P2) A related problem is the reinterpretation of data as bit patterns of non-floating-point type effected via equivalence statements such as

TYPE(INPT_RNEUTRPrototype) :: RNEUTR

INTEGER(KIND=KIND(1)), PRIVATE :: RNEUTRInt(1)

REAL(KIND=KIND(1.0d0)), PRIVATE :: RNEUTRDb1(1)

EQUIVALENCE (RNEUTR, HEXPCH, RNEUTRINT, RNEUTRDb1)

where storage space starting at the common block element *HEXPCH* is accessible by an integer or a double-precision array. No standard code analysis has been implemented in any of the current AD tools to determine whether all the operations performed on the reinterpreted bit patterns are benign (**P2a**: verify the absence of nontrivial operations). As mentioned before, an activation of elements of the equivalenced common block and an access of that modified common block through an array of a different type will lead to misalignment problems (**P2b**). This type of problem is extremely difficult to diagnose and, short of manually removing the *equivalence* and replacing the associated functionality in the source code, no easy remedy exists.

(P3) Another, similar problem is the use of the transfer intrinsic, as in the following line:

PRIMINSize = SIZE(TRANSFER(PRIMIN, PRIMINDbl)) ! Determine Block Size

It, too, implies a bit-pattern reinterpretation followed by taking the size of an array where the elements are the reinterpreted bit patterns. For each of these size values one would have to manually investigate (**P3a**) how exactly the size is used. A use like

CALL INPT_IOBaseReadINP(unit, PRIMINDbl, PRIMINSize)

suggests I/O operations through the equivalenced array. If we assume that the equivalenced block contains only double-precision elements, all uniformly redeclared as active, then the bit-pattern reinterpretation (depending on the data augmentation method used by the AD tool) may include with *size* also all derivative data storage, thereby causing misalignment for this I/O subroutine call (**P3b**). Bit-pattern reinterpretations of any sort make consistent automatic data augmentation much harder.

(P4) Another problem is data dependencies through the use of blank common blocks with varying definitions. For example, compare the following lines:

COMMON // TSAT1(49) *!TNV1..31*

COMMON // ISGIN(40), ISGOUT(40) *!SS1L..19*

COMMON // BIG(1) *!REEC..5*

These are inscrutable to automatic code analysis (**P4a**) as well as to most programmers unless they happen to be familiar with the code. The subsequent reference to *BIG* in

NPLACE=LOCUS+IOFFST-1

DO 100 I=1,LONG

W(I)=BIG(NPLACE+I)

hints at the fact that *BIG* is used as an array with more than 1 element but, as in the previously noted case, this fact is not reflected in the declaration syntax. What, if any, derivative data should be tracked between this and the other references to the blank common block storage remains unclear until a manual analysis of the code reveals the usage patterns (**P4b**).

(**P5**) There is a implicit assumption that the compiler will allocate storage to local variables following the Fortran 77 storage model with static allocation. This effectively promotes all variables to have global life span. Thus, a value assigned to a variable *v* that is local to subroutine *foo* during a given execution of *foo* can be accessed during the following execution of *foo*. The SAS code makes use of this effect in a number of places. Since Fortran 90 local

variables can be dynamically allocated on the stack, in order to enforce the older storage model, the SAS code must be compiled with the respective option (*-save* for Intel's ifort). Doing so, however, prevents certain compiler optimizations. Furthermore, the AD source transformation requires a similar modification (currently not implemented) for the dependency analysis overriding the actual scope information and giving all variables global life span. Eventually this leads to less-precise analysis results and unnecessary overhead for the adjoint computation. Instead of global promotion, the variables requiring the rescoping should be identified and declared with the Fortran *save* attribute.

Data passing. Because of hardware memory limitations an obfuscation such as the use of *BIG* in a blank common block discussed previously had to be accepted as a tradeoff for a reduced memory footprint of the program. The logic referring to *BIG* also hints at the (pseudo) address arithmetic performed using Fortran integers where *LOCUS* is some base address in memory and *IOFFST* an offset to that base address. This is done to move data from one memory region to another, where certain subroutines can then operate on it. Much like the previously discussed storage, merging this programming pattern obfuscates the data dependencies by funneling otherwise distinct data through the *BIG* array. This process is akin to low-level C-style passing of data with *memcpy* via pointers of type *void**. There is no syntactic hint about the alignment and therefore little information for any automatic transformation (**P6**) or, again, for most programmers not intimately familiar with the SAS code. Furthermore, the implementation of the (pseudo) address arithmetic restricted the code to 32-bit platforms, because Fortran integer arithmetic does not automatically reflect the different address length on a 64-bit system. Given the nonportable logic, it is a foregone conclusion that semantically correct AD data augmentation would be difficult to implement manually and virtually impossible for an automatic transformation.

In addition, the current implementation uses wrapper routines for system library functionality such as *malloc* and *free* that are provided only as compiler-specific extensions to the Fortran standard but not part of the standard and therefore not an integral part of the code analysis based on the Fortran memory model (**P7**). The introduction of dynamic memory concepts to the recent Fortran standards makes obsolete the need to refer to C functionality. Furthermore, the Fortran syntax provides more information to the AD tool. In particular, the restrictions on aliasing implied by the use of *allocatable* variables over pointer variables are beneficial for compiler optimization and source transformation (*P8*). Using *allocatable* variables also prevents memory leaks.

Control flow. The data flow reversal needed for adjoint computations implies a control flow reversal for the transformed code. The code exhibits numerous cases of unstructured control flow, such as the use of go to, early returns from loops or branches, and alternative *entry*. The ability to jump from one instruction to another is limited by the Fortran standard. Consequently, not every legitimate forward jump in the control flow graph has a legitimate reverse jump in the control flow graph with the same underlying structure (but reversed edges).

In contrast, structured control flow is free of such jumps. In a simplified view with structured control flow, every path taken through a loop or branch node in the control flow graph has to go through the corresponding endloop or endbranch node. This requirement permits control flow reversal by a control flow graph in the transformed code that is identical to the original control flow graph except for reversed edges. The concrete reversed path

through the graph at execution time is controlled by the same conditions as the forward path, and therefore only a minimum of information has to be recorded to reverse the concrete path.

Unstructured control flow for the AD-enabled adjoint computations requires an explicit recording of the path through the executed statements and a reversed replay for the data flow reversal to be syntactically permissible. This implies a significant overhead for the adjoint computation and a slow execution because it effectively removes optimizable loops and branches from the reversed control flow (**P9**) and leaves a flat set of statement sections whose execution pattern is unknown at compile time. Various compiler optimization techniques benefit from structured control flow as well.

We summarize the code features with suggested workarounds in Table 1. In the table, AD workaround indicates whether some intervention can make AD work without completely replacing the problematic idiom. Partial tool support indicates that some usage scenarios are handled but that incomplete coverage will require some more costly intervention in the use code if the usage falls outside of the covered patterns. Note that some of the problematic idioms include others: P2b includes P1c, and P4b includes P1b. The workaround suggested in P2a and P3a is an extensive manual check to assert the absence of the respective patterns in the code. Portability is meant to be among hardware platforms and compilers. Verification is meant here in a wider sense but also includes aspects of the typical verification and validation process. A particular aspect is the ability to assert facts about the use and update of model data by the executable statements in the program. For example, to verify that a given model setup uses exclusively “regime A” to update the temperature data, one would like to assert that the temperature field is assigned only in routine *foo,* whose logic can be checked to implement “regime A.” The use of *equivalence*, *memcpy* as well as the reliance on externally agreed-upon offsets to discriminate between the different data portions precludes simple and conclusive checks searching for syntactic references to the temperature portion of model data and allowing to make the assertions. Instead, the model data is accessed, copied, and copied back in ways that are hard to trace even manually. The corollary is the ease by which off-by-one index errors and offset mistakes can be introduced (undetectable by compiler or runtime checks) and the fact that these errors may remain undetected.

Table 1: Code idioms and suggested workarounds

	AD workaround	Negative impact on:						
		AD difficulty	SAS efficiency	Verification	Debugging	Portability	Standards	Maintainability
P1a: *equivalence* differentiable and passive data	automatic	X						
P1b: *equivalence* with incorrect array bounds	manual	X		X	X		X	
P1c: *equivalence* array with mixed-type common block	no							
P2a: check non-*real* operations done on bit patterns of *real* data	manual (costly)							
P2b: *equivalence* to non-*real* type yields misalignment of differentiable data	no			X	X	X	X	X
P3a: use of *size* after *transfer* on differentiable data	manual (costly)							
P3b: use of *transfer* of diffentiable data for I/O	no			X	X	X		
P4a: blank *common* block with mismatching definitions no	no							X
P4b: blank *common* block with single ”big” array of length 1	no			X	X	X		X
P5: enforce static allocation of all variables by compiler switch automatic	automatic	X	X					
P6: data passing through anonymous memory with pseudo address arithmetic (*IOFFST, LOCUS*)	no			X	X	X	X	X
P7: use of C-library calls *malloc* and *free* (instead of Fortran memory mgmt.)	no			X	X	X	X	X
P8: use of pointer where *allocatable* is appropriate.	partial support	X	X					X
P9: unstructured control flow via *goto* and *entry*	partial support	X	X					X

2.4 Code review: incremental modification vs incremental rewrite

Arguably, certain changes are unavoidable to enable AD. The complicated semantic transformations for adjoint computations only exacerbate the ill effects of the identified problems. A fitting analogy might be the code changes needed if a program that has been written over many years for sequential execution is to be parallelized. We note that none of the changes required for AD contradict good coding practices but instead contribute to the earlier-stated general goals for a redesign of parts of SAS. Two main groups of changes can be identified.

Redesign the data model. While the file formats used to initialize the data for a SAS model run and capture the output will have to remain invariant, the internal representation should be changed, and in the process the offending programming patterns should be completely eliminated. The following aspects should be considered for the data layout:

• Globally accessible data organized to achieve locality by physical properties of the engineering problem and functionality implemented in SAS permits easy tracking of data references. Either subroutines either directly access the global data implemented as module variables, or direct references to the global data are passed as arguments.

• An alternative data layout that is built on the fly as stack variables local to certain top-level subroutines will always imply more passing of data as subroutine arguments and more encapsulation of data in nested types and makes tracking data dependencies more difficult.

• Employing user-defined types, one can nest data structures. This nesting can be used to reflect locality of groups of data in the physical setup being modeled. While such nesting is plausible, however, it may not provide the locality needed by the computation. For example, one may want to have immediate access to all temperature data instead of first having to extract the temperature data buried in a data structure reflecting the physical assembly. Hierarchical structures can still be reflected in the temperature array by using multiple ranks. Globally accessible data allow for simpler (less-nested) data structures.

• For parallel execution the top-level structure in the data model should reflect the partitioning. As long as each process still executes sequentially, the above remarks in favor of globally accessible data still hold. In many cases multithreaded parallelism can also be implemented to operate on partitioned global data rather than thread-private data although perhaps then using few of the typical usages patterns supported by established tools.

• For dynamic memory allocation the use of *allocatable* is much preferred over *pointer* because it limits the possibility for aliasing. For performance and adjoint code generation it is highly desirable to allocate once and reuse over repeated local allocate/deallocate pairs.

• Allocation with constant array bounds, where possible, is preferred, in particular because it can be used to fix loop bounds. Compiler optimization and AD control flow reversal benefit from fixed array and loop bounds.

• Recursive data structures (e.g., linked lists) and other uses of pointers should be avoided unless required. The use of pointers poses problems for checkpointing (similar to problems known in the context of serialization) and the restoring of address values; both are needed for the data flow reversal.

Modify control flow and encapsulation. One should be able to rewrite the logic in such a way as to avoid unstructured control flow. Concerns with handling of unstructured control flow, the data model, and so forth should not apply to source code that does not contain operations that need differentiation. Therefore, it is helpful to extract the numerical core from the model code and analyze and transform only the core. Because of the nesting afforded by using module or subroutines with contains, one can group numerical functionality with initialization, debugging, and so on in a single syntactic envelope. The envelopes make it difficult to separate out the numerical core. In addition to commonly accepted rules for encapsulations, it is therefore desirable for AD to go a step further and encapsulate the common data, the numerical computations, and the ancillary logic in separate entities.

A modularization of the code that is enhanced by preprocessor directives should enable building different configurations by filtering out both unnecessary parts of the data model and the logic (calls to subroutines and their definitions).

Given the above, one should note that the overwhelming majority of the work comes with the data model redesign because it has a global effect on the source code and is more difficult to regression test. On the other hand, modifying the control flow is a local change that is comparatively easy to test for consistency with the unmodified source code.

Possible remedy: incremental modification. Incrementally modifying the existing source code while maintaining all existing functionality is an approach most suited to local changes. It may be possible to achieve the redesign in this way, and the advantage is the uninterrupted availability of all functionality of SAS. Given the need to modify the globally accessible data model, however, the required changes will have a wide scope, implying massive modifications throughout the source code. In other words, the ensuing changes are unlikely to be "incremental." Chances for introducing errors increase, and so does the effort to maintain all functionality. Because of this extra effort, the cost for making corrections as the data model is being changed is substantial.

Preferred remedy: incremental rewrite. Realizing the global impact of the data model one might instead start with designing a new data model. With this model one would then piecewise rewrite to match the new data model:

- Logic for initialization
- Logic for result outputs
- Individual physics components that make up SAS

Identifying the components, where there are not well separated from each other in the current SAS implementation, offers the chance to modularize the code in terms of both encapsulation and the preprocessor filtering suggested in the beginning of this section. After integrating each piece, the consistency of the output with the original SAS output should be verified to build up the prerequisite for the eventual switch over to the rewritten SAS code.

By designing the data model first and then rewriting the components, one has better granularity of the work units and can achieve a truly incremental approach. Experience suggests an overall lower effort and much improved tractability compared with the first approach. The design of the data model as the critical first step requires expert insight into the (abstract) data access patterns of the SAS components and ideally also the review by an AD expert.

2.5 Automatic differentiation of SHARP component UNIC

From the conclusions drawn from the effort in SASSYS, the application of AD methodology to the UNIC code was motivated as follows:

- Identification at a high level an eigenproblem that is solved in the UNIC code. Starting from the high-level description, which lends itself to a high-level adjoint formulation, we then identify a subproblem that warrants the use of AD. By restricting the application of AD to the subproblem, we expect to reduce the amount of UNIC source code to be transformed by the AD tool.
- The expectation that the Fortran source code in UNIC, having a more recent origin, was not containing any of the programming patterns that prohibited the application of AD to SASSYS.

While both points were met in principle and the application of AD was eventually successful, the path was not straightforward. The lessons learned will be applicable to continued work. The major phases of the AD application are outlined below.

Identify the source code representing the subproblem. In view of the high-level characterization of the relevant part of the computation as solving an eigenproblem using an iterative method, it is often not advisable to subject the entire source code to AD. The reasons include the possibility of using black-box libraries for the iterative solve steps and the efficiency gains obtainable by exploiting the high-level insight that allows an inexpensive adjoint formulation and use of the AD tools where necessary. Here, this means identifying the logic in UNIC that implicitly applies the system matrix to a given vector.

The top-level procedure (SN2NDz_MGS_ApplyA) was known, but in a static analysis included references to UNIC functionality and external libraries not needed for the proof of concept to be shown here for a benchmark problem. To suitably restrict the scope, therefore, we profiled the code in order to identify the subset of the UNIC source code relevant to the benchmark. While this represented the functional dependencies, a first round of testing clearly showed that the global data (module variables) imply additional dependencies, requiring the transformation of significantly more (a factor of 4) code to ensure the correct data setup.

Determine the data dependencies. Based on the results of the first round of testing, we again carried out benchmark profiling. The final amount of UNIC code to be differentiated was approximately 30K non-comment lines of Fortran. The additional code to be transformed in order to achieve consistent data initialization was approximately 90K non-comment lines of Fortran from a total of approximately 300 source files.

Devise a build procedure for the AD transformation. In order to achieve a consistent transformation, the AD tool must have a view of all the references to the data and logic that are to be transformed at the same time. The source files identified in the previous two steps stem from a variety of separately built UNIC components and therefore are not directly usable for the AD transformation. An AD-specific build procedure makes all the identified source files available (via symbolic links) to the AD tool in one step and enables linking with a top-level driver procedure modified for AD and other adaptations outlined in the next section.

Adapt the source code. While the UNIC source code base is comparatively clean, the introduction of certain recent Fortran language features can in some cases still pose inherent difficulties for the transformation. One notable example is the qualification of variables as pointer where a qualification as allocatable would be appropriate. The same assumptions allowing more aggressive compiler optimizations for variables with the latter qualification also permit a significantly simpler and more efficient adjoint transformation.

Other code modifications are caused by as-yet-unsupported language features such as the replacement of invocations of Fortran procedures via function pointers. Modifications to enable the computation of the derivatives for the subproblem include the injection of calls to hook methods on entry and exit of the top-level procedure (*SN2NDz_MGS_ApplyA*).

All changes are triggered by preprocessor directives and have been committed permanently to the UNIC source code repository.

Use library wrappers, drivers, verification. The benchmark test makes references to certain functionality provided by the PETSc library. Because the PETSc library was not differentiated, this functionality was wrapped to enable the derivative computation. The implementation of the hook methods to enable the computation of the inner product of the matrix vector product, as well as initialization and retrieval of the derivative values, is given as Fortran source. The wrappers, the hook methods, the driver for the transformed source code, and the makefiles for the transformation and linking of the transformed source code are permanently retained in a source code repository.

Initial tests indicate the correctness of the source transformations, implying the correctness of derivatives computed with AD.

3 Results and Discussion

The intermediate benefit of preparing SAS for algorithmic differentiation was an extensive review of code features and a set of recommendations for code modification. In the material above we have provided the main contents of this review, which can be used as a basis for future code development and standardization. This project could be undertaken in cooperation with developers of AD tools.

The changes made under either incremental modification or the incremental rewrite approach will have to be tested by regression and eventually validated. The timeline implied by the second (preferred) approach suggests an incremental buildup of regression tests corresponding to the integration of rewritten components. The numerically stable output values need to be identified, separated from inherently unstable outputs (subject to numerical noise), and assigned limit values for absolute and relative discrepancy. The obtained knowledge of the structure of the code and at least partial AD capability for the integrated components would be helpful here. In particular, elements of sensitivity analysis can be used to greatly speed the process of establishing precision thresholds and ranking parameters by stability and importance.

The task of validation here means partial validation that the computed results correspond to the externally established principles (such as physical conservation laws, or closeness to known exact solution). Such tests will also be easier to formulate on components of the system as they are integrated under the incremental rewrite approach.

```
new Weilandt =  0.100000000E+19
new MGS_Krylov_eTarget =     0.700
 AD output:    85.403252086713138        7.19880238E-02
I=     0 Norm = 3.331624935E+00 RT= 2.332137455E+00 AT= 1.869849136E-09
[GMRES]...Solving Ax=b with FGMRES with size       4470
[GMRES]...Absolute error   =   1.869849E-09
[GMRES]...Relative error   =   2.332137E+00
[GMRES]...Divergence error =   3.334957E+00
[GMRES]...Initial  error   =   3.331625E+00
 AD output:   -7.9329958062968631        7.59870186E-02
I=     0 Norm = 3.149056077E+00 RT= 2.332137455E+00 AT= 1.869849136E-09
 AD output:   -3.2985169842813904        7.99870118E-02
I=     1 Norm = 2.722483584E+00 RT= 2.332137455E+00 AT= 1.869849136E-09
 AD output:   0.28610342865643063        8.39860216E-02
I=     2 Norm = 2.137078052E+00 RT= 2.332137455E+00 AT= 1.869849136E-09
 AD output:    46.003694824369532        8.79850313E-02
I=     3 Norm = 2.137078052E+00 RT= 2.332137455E+00 AT= 1.869849136E-09
[SN2NDz]...MGS Rel Target  0.70E+00 Abs Target  0.19E-08 Reason   2 Iterations
=SN2NDz     0.0|005| 1.01593E-01| 5.6E-01|3.6E-01| 1.1E+01| 1.0E+18| T   0  0.640|
|  0   1|   0|      0|   0|   0|      0|   0|      0|      0|
Flux Dom Data=   0.4854
Flux Dom Data=   0.5940
Flux Dom Data=   0.6488
Flux Dom Fit Avg=   0.5761 RMS=   0.0679
-2 MGS Krylov eTarget =     0.700
```

Figure 2: UNIC differentiation diagnostics, partial output

At the current allocation of resources and with no significant code modification of SAS, capability for algorithmic differentiation cannot be achieved, and the preparation work performed for AD will be set aside as reference material to assist with the future code development (in context of AD capability or otherwise).

With the limited amount of effort redirected to differentiation of UNIC, we were able to achieve capability for direct differentiation. At this point, the effort to fully verify this capability is not finished. The interpretation of results beyond the algorithmic correctness of AD is still in process, and the diagnostics we are relying on (see an example of the output in Figure 2) are not easy to examine and interpret. Clearly, additional effort is required to attach this newly obtained capability to practical tasks related to improved performance and use of UNIC.

Overall, we maintain that, with additional effort required for codes with obsolete programming components, algorithmic differentiation remains a powerful and effective tool for code analysis. Once the technical difficulties of obtaining derivatives are resolved, it will drive the effort in advanced uncertainty analysis for models of high complexity and high dimension of uncertainty.

4 Summary

In our work in FY11-12, we performed an extensive analysis of the SHARP component code SAS, in the context of algorithmic differentiation. An intermediate result was a detailed (and otherwise not practically possible) annotation of the code for future development and standardization. While none of the mathematical (essential) components of the model prevented algorithmic differentiation, code organization and features in variables and memory management presented technical difficulties. We concluded that the capability for AD cannot be achieved without code modification. We have outlined two plans for possible code

improvement and argued that such effort will eventually become necessary for multiple reasons, many of them coming from wider context than just AD. We briefly identified future work in verification and validation of modified code as it is being developed.

Algorithmic differentiation of high-performance simulation codes was demonstrated (in the initial stages) on UNIC. This required comparatively less effort and is perhaps a better example of how we expect AD preparation process will work, given adequate resource allocation.

In the context of the NEAMS mission, our effort on implementing AD is a driving effort and one of the possible key elements that will enable advanced uncertainty analysis of codes previously inaccessible to any but the most basic (and computationally expensive!) uncertainty quantification techniques. We hope to continue the development work as a part of an independent effort or within practical code development and analysis tasks of NEAMS.

Acknowledgments

We thank Thomas Fanning and Mike Smith for their assistance with aspects of our work related to SAS and UNIC.

References

1. 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(4) pp. 1–36, 2008.
2. A. Griewank, "On Automatic Differentiation," Tech Report CRPC-TR89003, Center for Research on Parallel Computation, Rice University, 1989.
3. O. Roderick, M. Anitescu, and P. Fischer, "Polynomial Regression Approaches Using Derivative Information for Uncertainty Quantification," *Nuclear Science and Engineering*, 164(2) pp. 122–139, 2010.
4. B. Lockwood and M. Anitescu, "Gradient-Enhanced Universal Kriging for Uncertainty Propagation," *Nuclear Science and Engineering*, 2011.
5. O. Roderick, Z. Wang, and M. Anitescu, "Dimensionality Reduction for Uncertainty Quantification of Nuclear Engineering Models," *Transactions of the American Nuclear Society*, Hollywood, FL, 104, 2011.
6. F. Dunn and F. Prohammer, "SASSYS LMFFBR Systems Analysis Code," *Mathematics and Computers in Simulation*, 26(1) pp. 23–36, 1984.
7. M. Smith, D. Kaushik, A. Wollaber, W.S. Yang, and B. Smith, "Recent Research Progress on UNIC at Argonne National Laboratory," *International Conference on Mathematics*, Saratoga Springs, NY, 2009.
8. M. Anitescu, O. Roderick, Argonne National Laboratory, unpublished information, 2011.
9. H. Edwards et al., "Nuclear Energy Advanced Modeling and Simulation Waste Integrated Performance and Safety Codes (NEAMS Waste IPSC), Verification and Validation Plan, Tech. report, 2011.

10. A. Griewank, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, SIAM, 2000.

11. M. Alexe, O. Roderick, J. Utke, M. Anitescu, P. Hovland, and T. Fanning, "Automatic Differentiation of Codes in Nuclear Engineering Applications," Tech. rep, ANL/MCS-TM-310, 2009.

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue, Bldg. 240
Argonne, IL 60439

www.anl.gov