

Combining Automatic Differentiation Methods for High Dimensional Nonlinear Models

James A. Reed, Jean Utke, and Hany S. Abdel-Khalik

Abstract Earlier work has shown that the efficient subspace method can be employed to reduce the effective size of the input data stream for high dimensional models when the effective rank of the first order sensitivity matrix is orders of magnitude smaller than the size of the input data. In this manuscript, the method is extended to handle nonlinear models, where the evaluation of higher order derivatives is important but also challenging because the number of derivatives increases exponentially with the size of the input data streams. A recently developed hybrid approach is employed to combine reverse mode automatic differentiation to calculate first order derivatives and perform the required reduction in the input data stream followed by forward mode automatic differentiation to calculate higher order derivatives with respect only to the reduced input variables. Three test cases illustrate the viability of the approach.

Key words: reverse mode, higher-order derivatives, low-rank approximation

1 Introduction

As is the case in many numerical simulations in science and engineering, one can use derivative information to gain insight into the model behavior. Automatic differentiation (AD) [7] provides a means to efficiently and accurately compute such derivatives to be used, for example, in sensitivity analysis, uncertainty propagation, and design optimization. The basis for AD is the availability of a program that implements the model as source code. Transforming or reinterpreting the source code

James A. Reed, Hany S. Abdel-Khalik
Dept. of Nuclear Engineering, North Carolina State University, Raleigh, NC, USA, {jareed3 |
abdelkhalik}@ncsu.edu

Jean Utke
Argonne National Laboratory / The University of Chicago, IL, USA, utke@mcs.anl.gov

enables the derivative computation. Given the complexity of the numerical simulations the derivative computation can remain quite costly, despite the efficiency gains made possible by AD techniques.

Exploiting model properties that are known at a higher mathematical level but are not easily recognizable at the source code level in an automatic fashion is a major factor for improving the efficiency of derivative based methods. Problems in nuclear engineering provide a good example for such higher-level properties. Detailed nuclear reactor simulations involve high-dimensional input and output streams. It is, however, known that the effective numerical rank r of such models is typically much lower than the size of the input and output streams would naively suggest. By reducing the the higher-order approximation of the model to r (pseudo) variables one can significantly reduce the approximation cost while maintaining reasonable approximation errors. This approach, the efficient subspace method (ESM), is discussed in Sect. 2. The implementation with AD tools is described in Sect. 3 and the paper closes with three test cases in Sect. 4.

2 Methodology

For simplicity we begin with constructing a low rank approximation to a matrix operator. Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be the unknown matrix and the operator provide matrix vector products with \mathbf{A} and \mathbf{A}^T . The following steps provide a low rank approximation of \mathbf{A} are as follows:

1. Form k matrix vector products $\mathbf{y}^{(i)} = \mathbf{A}\mathbf{x}^{(i)}$, $i = 1, \dots, k$ for randomly chosen $\mathbf{x}^{(i)}$
2. QR factorize the matrix of responses: $[\mathbf{y}^{(1)} \dots \mathbf{y}^{(k)}] = \mathbf{Q}\mathbf{R} = [\mathbf{q}^{(1)} \dots \mathbf{q}^{(k)}] \mathbf{R}$
3. Determine the effective rank r :
 - a. Choose a sequence of k random Gaussian vectors $\mathbf{w}^{(i)}$
 - b. Compute $\mathbf{z}^{(i)} = (\mathbf{I} - \mathbf{Q}\mathbf{Q}^T)\mathbf{A}\mathbf{w}^{(i)}$
 - c. Test for any i if $\|\mathbf{z}^{(i)}\| > \varepsilon$ then increment k and go back to step 1 else set $r := k$ and continue.
4. Calculate $\mathbf{p}^{(i)} = \mathbf{A}^T \mathbf{q}_{(i)}$ for all $i = 1, \dots, k$
5. Using the $\mathbf{p}_{(i)}$ and $\mathbf{q}_{(i)}$ vectors, a low rank approximation of the form $\mathbf{A} = \mathbf{U}\mathbf{S}\mathbf{V}^T$ can be calculated as show in the appendix of [1]

It was shown in [8] that with at least $1 - 10^{-k}$ probability, one can determine a matrix \mathbf{Q} of rank r such that the following error criterion is satisfied

$$\|(\mathbf{I} - \mathbf{Q}\mathbf{Q}^T)\mathbf{A}\| \leq \varepsilon / (10\sqrt{2/\pi})$$

where ε is the user specified error allowance.

In real applications, these ideas can be applied by replacing the matrix operator with a computational model. Let the computational model of interest be described by a vector-valued function $\mathbf{y} = \Theta(\mathbf{x})$, where $\mathbf{y} \in \mathbb{R}^m$ and $\mathbf{x} \in \mathbb{R}^n$. The goal is to

compute all derivatives of a given order by reducing the dimensions of the problem and thus reducing the computational and storage requirements. First we consider the case $m = 1$. A (possibly non-linear) function $\Theta(\mathbf{x})$ can be expanded around a reference point \mathbf{x}_0 . It was shown in [2] that an infinite Taylor-like series expansion may be written as follows (without loss of generality, assume $\mathbf{x}_0 = \mathbf{0}$ and $\Theta(\mathbf{x}_0) = 0$)

$$\Theta(\mathbf{x}) = \sum_{k=1}^{\infty} \sum_{j_1, \dots, j_l, \dots, j_k=1}^n \psi_1(\beta_{j_1}^{(k)T} \mathbf{x}) \dots \psi_l(\beta_{j_l}^{(k)T} \mathbf{x}) \dots \psi_k(\beta_{j_k}^{(k)T} \mathbf{x}) \quad (1)$$

where the $\{\psi_l\}_{l=1}^{\infty}$ can be any kind of scalar functions. The outer summation over the variable k goes from 1 to infinity. Each term represents one order of variation, e.g. $k = 1$ represents the first order term; $k = 2$, the second order terms. For the case of $\psi_l(\theta) = \theta$, the k^{th} term reduces to the k^{th} term in a multi-variable Taylor series expansion. The inside summation for the k^{th} term consists of k single valued functions $\{\psi_l\}_{l=1}^{\infty}$ that are multiplying each other. The arguments for the $\{\psi_l\}_{l=1}^{\infty}$ functions are scalar quantities representing the inner products between the vector \mathbf{x} and n vectors $\{\beta_{j_l}^{(k)}\}_{j_l=1}^n$ which span the parameter space. The superscript (k) implies that a different basis is used for each of the k -terms, i.e. one basis is used for the first-order term, another for the second-order term and so on.

Any input parameter variations that are orthogonal to the range formed by the collection of the $\{\beta_{j_l}^{(k)}\}$ vectors will not produce changes in the output response, i.e. the value of the derivative of the function will not change. If the $\{\beta_{j_l}^{(k)}\}$ vectors span a subspace of dimension r as opposed to n ($\text{span}\{\beta_{j_l}^{(k)}\} = \mathbb{R}^r$), then the effective number of input parameters can be reduced from n to r . The mathematical range can be determined by using only first-order derivatives. Differentiating (1) with respect to \mathbf{x} gives

$$\nabla \Theta(\mathbf{x}) = \sum_{k=1}^{\infty} \sum_{j_1, \dots, j_l, \dots, j_k=1}^n \left(\psi'_l(\beta_{j_l}^{(k)T} \mathbf{x}) \beta_{j_l}^{(k)} \prod_{i=1, i \neq l}^k \psi_i(\beta_{j_i}^{(k)T} \mathbf{x}) \right) \quad (2)$$

where $\psi'_l(\beta_{j_l}^{(k)T} \mathbf{x}) \beta_{j_l}^{(k)}$ is the derivative of the term $\psi_l(\beta_{j_l}^{(k)T} \mathbf{x})$. We can reorder (2) to show that the gradient of the function is a linear combination of the $\{\beta_{j_l}^{(k)}\}$ vectors

$$\nabla \Theta(\mathbf{x}) = \sum_{k=1}^{\infty} \sum_{j_1, \dots, j_l, \dots, j_k=1}^n \chi_{j_l}^{(k)} \beta_{j_l}^{(k)} = \begin{bmatrix} \beta_1^{(1)} & \dots & \beta_{j_l}^{(k)} & \dots \end{bmatrix} \begin{bmatrix} \chi_1^{(1)} \\ \vdots \\ \chi_{j_l}^{(k)} \\ \vdots \end{bmatrix} = \mathbf{B} \boldsymbol{\chi}$$

where

$$\chi_{j_l}^{(k)} = \psi'_l(\beta_{j_l}^{(k)T} \mathbf{x}) \prod_{i=1, i \neq l}^k \psi_i(\beta_{j_i}^{(k)T} \mathbf{x})$$

In a typical application, the \mathbf{B} matrix will not be known beforehand. It is only necessary to know the range of \mathbf{B} which can be accomplished using the rank finding algorithm, see above. After determining the effective rank, it can be seen that the function only depends on r effective dimensions and can be reduced to simplify the calculation. The reduced model only requires the use of the subspace that represents the range of \mathbf{B} , of which there are infinite possible bases.

This concept is now expanded to a vector-valued model. The q^{th} response $\Theta_q(\mathbf{x})$ of the model and its derivative $\nabla\Theta_q(\mathbf{x})$ can be written just like (1) and (2) with an additional index q in the vectors $\{\beta_{j_i,q}^{(k)}\}$. The active subspace of the overall model must contain the contributions of each individual response. The matrix \mathbf{B} will contain the $\{\beta_{j_i,q}^{(k)}\}$ vectors for all orders and responses. To determine a low rank approximation, a pseudo response Θ_{pseudo} will be defined as a linear combination of the m responses:

$$\Theta_{pseudo}(\mathbf{x}) = \sum_{q=1}^m \gamma_q \sum_{k=1}^{\infty} \sum_{j_1, \dots, j_k=1}^n \psi_1(\beta_{j_1,q}^{(k)T} \mathbf{x}) \dots \psi_l(\beta_{j_l,q}^{(k)T} \mathbf{x}) \dots \psi_k(\beta_{j_k,q}^{(k)T} \mathbf{x}) \quad (3)$$

where γ_q are randomly selected scalar factors. The gradient of the pseudo response is:

$$\nabla\Theta_{pseudo}(\mathbf{x}) = \sum_{q=1}^m \gamma_q \sum_{k=1}^{\infty} \sum_{j_1, \dots, j_k=1}^n \left(\psi'_l(\beta_{j_l,q}^{(k)T} \mathbf{x}) \beta_{j_l,q}^{(k)} \prod_{i=1, i \neq l}^k \psi_i(\beta_{j_i,q}^{(k)T} \mathbf{x}) \right)$$

Calculating derivatives of the pseudo response as opposed to each individual response provides the necessary derivative information while saving considerable computational time for large models with many inputs and outputs.

3 Implementation

In this section we discuss the rationale for the specific AD approach, tool independent concerns and some aspects of applying the tools to the problem scenario.

3.1 Gradients with OpenAD

The numerical model Θ has the form $\mathbf{y} = \Theta(\mathbf{x})$, where $\mathbf{y} \in \mathbb{R}^m$ is the output and $\mathbf{x} \in \mathbb{R}^n$ the input vector. No additional information regarding Θ is required except for the program P implementing Θ . Following (3) we define the pseudo response $\tilde{\mathbf{y}}$ as the weighted sum

$$\tilde{\mathbf{y}} = \sum_{i=1}^m \gamma_i \mathbf{y}_i \quad (4)$$

This requires a change in P but is easily done at a suitable top level routine. The source code (Fortran) for the modified program \tilde{P} becomes the input to OpenAD [10] which facilitates the computation of the gradient $\nabla\tilde{y}$ using reverse mode source transformation. The overall preparation of the model and the first driver was done following the steps outlined in [11].

The source code of P used for the test cases described in Sect. 4 exhibited some of the programming constructs known to be obstacles for the application of source transformation AD. Among them are the use of `equivalence` especially for the initialization of `common` blocks. The idiom there was to equivalence an array of length 1 with the first element in the common block. Then the length-1 array was used to access the entire common block via subscript values greater than 1. The Fortran standard requires that a subscript value shall be within the bounds for its dimension but typically this cannot be verified at compile time and therefore this non-standard is tacitly accepted. It has since appeared that the same pattern is also used in other nuclear engineering models making this problem more interesting than it would be for just for a single occurrence.

OpenAD uses *association by address* [5], that is an active type, as the means of augmenting the original program data to hold the derivative information. The usual activity analysis would ordinarily trigger the redeclaration of only a subset of common block variables. Because the access of the common block via the array enforces a uniform type for all common block variables to maintain proper alignment, all common block variables had to be activated. Furthermore, because the equivalence construct applied syntactically only to the first common block variable, the implicit equivalence of all other variables cannot be automatically deduced and required a change of the analysis logic for OpenAD to maintain alignment by conservatively overestimating the active variable set.

Superficially this may seem a drawback of the association by address. The *association by name* [5], used in other AD source transformation tools will not fare better though. Shortening the corresponding loop for the name-associated and equivalenced derivative-carrying array is difficult for interspersed passive data and therefore one will resort to the same alignment requirement.

Once the source transformation succeeds a suitable driver logic has to be written to accommodate the steps needed for k evaluations of the gradient $\nabla\tilde{y}^{(j)}$ using random weights $\gamma_i^{(j)}$ and randomly set inputs $x_i^{(j)}$. The k gradients form the columns of

$$\mathbf{G} = \left[\nabla\tilde{y}^{(1)}, \dots, \nabla\tilde{y}^{(k)} \right]$$

The effective rank of \mathbf{G} will be found via the following modified version of RFA, cf. Sect. 2. A singular value decomposition (SVD) of \mathbf{G} is computed in the form $\mathbf{G} = \mathbf{U}\mathbf{S}\mathbf{V}^T$, with $\mathbf{U} \in \mathbb{R}^{n \times p}$, $\mathbf{V} \in \mathbb{R}^{k \times p}$, $\mathbf{S} = \text{diag}(\sigma_1, \dots, \sigma_p)$, and $\sigma_1 > \dots > \sigma_p \geq 0$ the singular values of \mathbf{G} . We successively drop the smallest singular values $\sigma_p, \sigma_{p-1}, \dots, \sigma_{p-l}$, compute $\mathbf{G}_{p-l} = \mathbf{U}\mathbf{S}_{p-l}\mathbf{V}^T$ and test

$$\frac{\|\mathbf{G} - \mathbf{G}_{p-l}\|}{\|\mathbf{G}\|} < \varepsilon \quad (5)$$

for a given ε . The last $\mathbf{G}_{p-l} \equiv \tilde{\mathbf{G}}$ satisfying (5) has rank $r = p - l$. Next, $\tilde{\mathbf{G}}$ is QR factorized

$$\tilde{\mathbf{G}} = \mathbf{QR} = [\mathbf{Q}_r \mathbf{Q}_2] \mathbf{R}$$

where the submatrix $\mathbf{Q}_r \in \mathbb{R}^{n \times r}$ contains only the first r columns of \mathbf{Q} . The columns of \mathbf{Q}_r are used to define the (reduced) pseudo inputs.

$$\tilde{\mathbf{x}} = \mathbf{Q}_r^T \mathbf{x}$$

Because of orthogonality we can simply prepend to the original program P logic implementing $\mathbf{x} = \mathbf{Q}_r \tilde{\mathbf{x}}$ to have the $\tilde{\mathbf{x}}$ as our new reduced set of input variables for which derivatives will be computed. Similar to (4), this is easily done by adding code in a suitable top level routine yielding $\hat{P}(\tilde{\mathbf{x}}) = \mathbf{y}, \hat{P} : \mathbb{R}^r \mapsto \mathbb{R}^m$.

3.2 Higher Order Derivatives with Rapsodia

Rapsodia [4] is used to compute all derivative tensor elements up to order o

$$\left[\frac{\partial^o y_i}{\partial \tilde{x}_1^{o_1} \dots \partial \tilde{x}_r^{o_r}} \right], \quad \text{with multi-index } \mathbf{o}, \text{ where } o = |\mathbf{o}| = \sum_{k=1}^r o_k, \quad (6)$$

for \hat{P} following the interpolation approach in [6] supported by Rapsodia, see also [3].

Rapsodia is in principle based on operator overloading for the forward propagation of univariate Taylor polynomials. All other operator overloading based AD tools have overloaded operators that are hand-coded, operate on Taylor coefficient arrays with variable length in loops with variable bounds to accommodate the derivative orders and numbers of directions needed by the application. In contrast, Rapsodia generates on demand a library of overloaded operators for a specific number of directions and a specific order. Thus, at compile time, the loops are already represented in (partially) unrolled code along with a fixed (partially flat) data structure that provides more freedom for compiler optimization. Because of the overall assumption that r , the reduced input dimension, is much smaller than m the higher order derivative computation in forward mode is feasible and appropriate.

Because overloaded operators are triggered by using a special (active) type for which they are declared it now appears as a nice confluence of features that OpenAD for the gradient computation already does the data augmentation via association by address, i.e. via an active type, and therefore the assumption could be made that one merely has to change the OpenAD active type to a Rapsodia active type to use the operator overloading library. The following features of the OpenAD type change already undertaken for Sect. 3.1 can (partially) be reused.

selective type change based on activity analysis: Here the main difference to Sect. 3.1 is the change of inputs from \mathbf{x} to $\tilde{\mathbf{x}}$ and conversely $\tilde{\mathbf{y}}$ to \mathbf{y} . This merely requires

```

subroutine foo(a,b,c)    real :: d, t2; type(active):: e, f t1
  type(active)::a,c    !...
  real::b                call cvrt_p2a(c,t1); call cvrt_a2p(d,t2)
  !....                  call foo(t1,t2,f)
end subroutine           call cvrt_a2p(t1,c); call cvrt_p2a(t2,d)

```

Fig. 1 Passive \leftrightarrow active type change conversions $\text{cvrt}_{\{p2a|a2p\}}$ for a subroutine call $\text{foo}(d, e, f)$ made by OpenAD for applying a Rapsodia-generated library (shortened names, active variables underlined).

changing the pragma declarations identifying the dependent and independent program variables in the top level routine.

type conversion for changing activity patterns in calls: The activity analysis intentionally does not yield matching activity signatures for all calling contexts of any given subroutine. Therefore, for a subroutine $\text{foo}(a, b, c)$, the formal parameters a, c may be determined as active while b remains passive. For a given calling context $\text{call foo}(d, e, f)$ it may be that the type of the actual parameter d is passive or e is active in which case pre and post conversion calls to a type-matching temporary may have to be generated, see an illustration in Fig. 1.

default projections to the value component: The type change being applied to the program variables, arithmetic and I/O statements referencing active variables are adapted to access the value component of the active type to replicate the original computation.

These portions are implemented in the `TypeChange` algorithm stage within the `xaifBooster` component of OpenAD. Of course, the last feature prevents triggering the overloaded operators and the value component access needs to be dropped from the transformation. On the other hand, as in most operator overloading tools there is, as a safety measure, no assignment operator or implicit conversion from active types to the passive floating point types. Therefore, assignment statements to passive left-hand sides need to retain the value component access in the right-hand-side expressions. These specific modifications were implemented in OpenAD's post processor with a `--overload` option. While manual type change was first attempted it quickly proved a time intensive task even on the moderately sized nuclear engineering source code in particular because of the many I/O statements that would need adjustments and the fact that the Fortran source code given in fixed format made simple editor search and replaces harder. Therefore this manual attempt was abandoned and this modification of the OpenAD source transformation capabilities proved useful.

Given the type change transformation, the tensors in (6) are computed with Rapsodia. The first order derivatives in terms of the \mathbf{x} rather than the $\tilde{\mathbf{x}}$ are recovered as follows.

$$\frac{\partial y_i}{\partial x_j} = \sum_{k=1}^r \underbrace{\frac{\partial y_i}{\partial \tilde{x}_k}}_{\in \tilde{J}} \frac{\partial \tilde{x}_k}{\partial x_j} = \sum_{k=1}^r \frac{\partial y_i}{\partial \tilde{x}_k} q_{jk}$$

In terms of the Jacobian this is $\mathbf{J} = \mathbf{Q}_r \tilde{\mathbf{J}}$. Similarly for second order one has

$$\frac{\partial^2 y_i}{\partial x_j \partial x_g} = \sum_{k,l} \underbrace{\frac{\partial^2 y_i}{\partial \tilde{x}_k \partial \tilde{x}_l}}_{\in \tilde{\mathbf{H}}_i} q_{jk} q_{gl}$$

which in terms of the Hessian \mathbf{H}_i for i -th output y_i is $\mathbf{H}_i = \mathbf{Q}_r \tilde{\mathbf{H}}_i \mathbf{Q}_r^T$. The o -th order derivatives are recovered by summing over the multi-indices \mathbf{k}

$$\frac{\partial^o y_i}{\partial x_{j_1} \dots \partial x_{j_o}} = \sum_{|\mathbf{k}|=o} \frac{\partial^o y_i}{\partial \tilde{x}_{k_1} \dots \partial \tilde{x}_{k_o}} \prod_{l=1}^o q_{j_l k_l}$$

For all derivatives in increasing order, products of the of the q_{jk} can be incrementally computed.

4 Test Cases

Simple scalar-valued model We consider an example model given as

$$y = \mathbf{a}^T \mathbf{x} + (\mathbf{b}^T \mathbf{x})^2 + \sin(\mathbf{c}^T \mathbf{x}) + \frac{1}{1 + e^{\mathbf{d}^T \mathbf{x}}}$$

where vectors $\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{c}$, and $\mathbf{d} \in \mathbb{R}^n$. The example model is implemented in a simple subroutine named `head` along with a `driver` main program that calls `head` and is used to extract the derivatives. Then `head` was transformed with OpenAD to compute the gradient of y with respect to the vector \mathbf{x} .

A Python script was written to execute the subspace identification algorithm with the compiled executable code. The script takes a guess k for the effective rank and runs the code for k random input vectors \mathbf{x} . Within the Python script, the responses are collected into a matrix \mathbf{G} . Following the algorithm, the singular values of \mathbf{G} are found and reduced versions of \mathbf{G} are calculated until the effective rank r is determined. A QR decomposition is then performed on the reduced matrix and the first r columns of \mathbf{Q} are written to a file to be used as input to the Rapsodia code. Using the model above with $n = 50$ and random input vectors with 8 digits of precision for $\mathbf{a}, \mathbf{b}, \mathbf{c}$, and \mathbf{d} , with $\varepsilon = 10^{-6}$, the effective rank was found to be $r = 3$.

The `driver` is then modified for the use with Rapsodia and the library is generated with the appropriate settings for the order and the number or directions. For first order this is simply the number of inputs. Once the derivatives $\left(\frac{dy}{d\tilde{x}}\right)$ are calculated, the full derivatives can be reconstructed by multiplying the Rapsodia results by the \mathbf{Q}_1 matrix used as input. Using an effective rank of $r = 3$ and therefore a \mathbf{Q}_1 matrix of dimension 50×3 , the reconstructed derivatives were found to have relative errors on the order of 10^{-13} compared to results obtained from an unreduced Rapsodia calculation.

Derivative Order	Unreduced Directions	Reduced Directions	Relative Error
1	50	3	10^{-13}
2	1275	6	10^{-12}
3	22,100	10	10^{-6}

Table 1 Comparison of number of required directions for unreduced and the reduced model together with the relative error

Using Rapsodia to calculate second order derivatives simply involves changing the derivative order to $o = 2$ and recompiling the code. The output can then be constructed into a matrix $\tilde{\mathbf{H}}$ of size $r \times r$ and the full derivatives can be recovered by $\mathbf{Q}_1 \tilde{\mathbf{H}} \mathbf{Q}_1^T$ which results in a $n \times n$ symmetric matrix. When the second order derivatives are calculated for the example above, there are only 6 directions required for an effective rank of 3 as opposed to 1275 directions for the full problem. The relative errors of the reduced derivatives are on the order of 10^{-12} .

Third order derivatives were also calculated using this example. The unreduced problem would require 22,100 directions while the reduced problem only requires 10. Relative errors were much higher for this case but still at a reasonable order of 10^{-6} . The relative errors for each derivative order are summarized in Table 1.

Simple Vector-valued model Problems with multiple outputs require a slightly differing approach when determining the subspace. First we consider

$$y_1 = \mathbf{a}^T \mathbf{x} + (\mathbf{b}^T \mathbf{x})^2; y_2 = \sin(\mathbf{c}^T \mathbf{x}) + (1 + e^{-\mathbf{d}^T \mathbf{x}})^{-1}; y_3 = (\mathbf{a}^T \mathbf{x})(\mathbf{e}^T \mathbf{x}); y_4 = 2e^{\mathbf{f}^T \mathbf{x}}; y_5 = (\mathbf{d}^T \mathbf{x})^{-3}$$

Following (4) we compute the pseudo response $\tilde{\mathbf{y}}$ in the modified version of the `head` routine implementing the above example model with randomly generated factors γ_i that are unique for each execution of the code. The computed gradient is of $\tilde{\mathbf{y}}$ with respect to \mathbf{x} . Then, following the same procedure as before, the subspace identification script was run for $n = 50$ and $\varepsilon = 10^{-6}$. The effective rank was found to be $r = 5$ and we found for similar accuracy the reduction of directions needed for the approximation from 250 to 5, 6375 to 75 and 110500 to 175 for first up to third order, respectively.

MATWS A more realistic test problem was done with the MATWS (a subset of SAS4A [9]) Fortran code for nuclear reactor simulations. Single channel calculations were performed using as inputs the axial expansion coefficient, Doppler coefficient, moderator temperature coefficient, control rod driveline, and core radial expansion coefficient. The outputs of interest are temperatures within the channel in the coolant, the structure, the cladding, and the fuel. This gives a 4×5 output for the first order derivatives and 15 and 35 directions for 2nd and 3rd order, respectively. After applying the subspace identification algorithm, it was found that the effective rank was $r = 3$, giving 6 and 10 directions for 2nd and 3rd order derivative calculations. Reductions and relative approximation errors are shown in Table 2.

This manuscript has presented a new approach to increase the efficiency of automatic differentiation when applied to high dimensional nonlinear models where

Derivative Order	Unreduced Directions	Reduced Directions	Relative Error
1	5	3	0.00625
2	15	6	0.038

Table 2 Comparison of number of required directions for unreduced and the reduced model together with the relative error

high order derivatives are required. The approach identifies few pseudo input parameters and output responses which can be related to the original parameters and responses via simple linear transformations. The AD is then applied to the pseudo variables which results in significant computational savings.

Acknowledgements This work was supported by the U.S. Department of Energy, under contract DE-AC02-06CH11357.

References

1. Abdel-Khalik, H.: Adaptive core simulation. Ph.D. thesis (2004). URL <http://books.google.com/books?id=5moolOgFZ84C>
2. Bang, Y., Abdel-Khalik, H., Hite, J.M.: Hybrid reduced order modeling applied to nonlinear models. *International Journal for Numerical Methods in Engineering* (to appear)
3. Charpentier, I., Utke, J.: Rapsodia: User manual. Tech. rep., Argonne National Laboratory. Latest version available online at <http://www.mcs.anl.gov/Rapsodia/userManual.pdf>
4. Charpentier, I., Utke, J.: Fast higher-order derivative tensors with Rapsodia. *Optimization Methods Software* **24**(1), 1–14 (2009). DOI 10.1080/10556780802413769
5. Fagan, M., Hascoët, L., Utke, J.: Data representation alternatives in semantically augmented numerical models. In: *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, pp. 85–94. IEEE Computer Society, Los Alamitos, CA, USA (2006). DOI 10.1109/SCAM.2006.11
6. Griewank, A., Utke, J., Walther, A.: Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation* **69**, 1117–1130 (2000)
7. Griewank, A., Walther, A.: *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd edn. No. 105 in *Other Titles in Applied Mathematics*. SIAM, Philadelphia, PA (2008). URL <http://www.ec-securehost.com/SIAM/OT105.html>
8. Halko, N., Martinsson, P.G., Tropp, J.A.: Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review* **53**(2), 217–288 (2011). DOI 10.1137/090771806. URL <http://link.aip.org/link/?SIR/53/217/1>
9. SAS4A: <http://www.ne.anl.gov/codes/sas4a/>
10. Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., Wunsch, C.: OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes. *ACM Transactions on Mathematical Software* **34**(4), 18:1–18:36 (2008). DOI 10.1145/1377596.1377598
11. Utke, J., Naumann, U., Lyons, A.: OpenAD/F: User Manual. Tech. rep., Argonne National Laboratory. Latest version available online at <http://www.mcs.anl.gov/OpenAD/openad.pdf>