Comput Geosci (2014) 18:1009–1022
DOI 10.1007/s10596-014-9441-z

ORIGINAL PAPER

A backward automatic differentiation framework for reservoir simulation

Xiang Li · Dongxiao Zhang

Received: 29 November 2013 / Accepted: 26 August 2014 / Published online: 17 September 2014

Abstract In numerical reservoir simulations, Newton's method is a concise, robust and, perhaps the most commonly used method to solve nonlinear partial differential equations (PDEs). However, as reservoir simulators incorporate more and more physical and chemical phenomena, writing codes that compute gradients for reservoir simulation equations can become quite complicated. This paper presents an automatic differentiation (AD) framework that is specially designed for simplifying coding and simultaneously maintaining computational efficiency. First a parse tree for a mathematical expression is built and evaluated with the backward mode AD, and then the derivatives with respect to the expression's arguments are transformed to derivatives with respect to the PDE's independent variables. The first stage can be realized either by runtime polymorphism to gain higher flexibility or by compile-time polymorphism to gain faster execution speed; the second stage is realized by linear combinations of sparse vectors, which can be accelerated by recording the target column indices. The AD framework has been implemented in an in-house reservoir simulator. Individual tests on some complex mathematical expressions were carried out to compare the speed of the manual implementation, the runtime polymorphic implementation and the compile-time polymorphic implementation of the differentiation. Then the performance of the three was analyzed in complete simulations. These cases indicate that the proposed approach has good efficiency and is applicable to reservoir simulations.

X. Li
College of Engineering, Peking University, Room 1009, Taipingyang Building, Peking University, Beijing, China
e-mail: lixiangcn@pku.edu.cn

D. Zhang (✉)
College of Engineering, Peking University, Building 60, Yannanyuan, Peking University, Beijing, China
e-mail: dxz@pku.edu.cn

Keywords Reservoir simulation · Automatic differentiation · Backward mode · Expression template

Mathematics Subject Classifications (2010) 65F02 · 68W02

1 Introduction

Reservoir simulation serves as a primary tool for quantitative reservoir management. The status of reservoirs, wells, and ground facilities are governed by nonlinear equations and are solved after these equations are discretized in time and in space. Explicit schemes often have numerical instability in solving these types of problems, unless small time steps are used, which lead to unacceptably slow simulation speeds. To ensure stability under larger time steps, a certain level of implicitness is needed. Newton's method is then used to solve the implicit equations; hence the residual vector's firstorder gradients with respect to the implicit unknowns are needed to assemble the Jacobian matrix. Writing analytical differentiation code by hand, known as "hand differentiation (HD)" is tedious and error prone, especially when the simulator needs to integrate many complicated models to model the recovery processes more accurately. Numerical differentiation (ND) can easily generate gradients for complicated and even black-box models, but it suffers from precision and efficiency problems [32]. In fact, HD is still the most common approach in commercial simulators, because manually optimized code produces the best computational performance. However, it takes a large human effort to linearize a long and deep expression. Furthermore, the developer may have to keep duplicate code implementations of the same mathematical model if the simulator contains different types of reservoir models, because the implicit unknown sets may be different. As a result, the simulation code becomes difficult to maintain.

An alternative choice for generating gradients is automatic differentiation (AD), the basic principle of which is the chain rule. Compared with ND, AD offers perfect accuracy up to machine precision; compared with HD, AD requires less human labor. Early research and implementations of AD focused on the forward mode [4, 31] in which the values and derivatives are calculated simultaneously. Forward mode is more convenient for programming, but less suitable for producing partial derivatives for "multiple input single output" functions, which are the usual cases in reservoir simulations. A complicated physical or chemical model often has a long list of arguments. Let n be the dimension of the argument vector $\mathbf{x}$; "work(g)" be the computational work of evaluating $g(\mathbf{x})$; and "work(g, ∇g)" be the computational work of evaluating $g(\mathbf{x})$ and its derivatives with respect to $\mathbf{x}$. The work ratio of forward mode satisfies an inequality: $\frac{\text{work}(g,\nabla g)}{\text{work}(g)} \geq 1 + \frac{n}{c}$ [13], where c is a positive constant $\left(\text{a special case is } c = 2 \text{ when } g(\mathbf{x}) = \prod_{i=1}^{n} x_i\right)$, so the additional computational work grows linearly with n. However, Wolfe [32] pointed out that if care is taken in HD code, the work ratio is usually around 1.5, and rarely exceeds 2; Baur and Strassen [3] proved that the nonscalar complexity of $\{g, \nabla g\}$, with "nonscalar complexity" defined as the minimal number of nonscalar multiplications and divisions sufficient to compute a set of functions [8], can be reduced to lower than three times of the nonscalar complexity of $\{g\}$.

While the forward mode could be far from optimal when the argument list is long, the other category of AD algorithms—the backward mode, also called the reverse mode [23, 25], is independent of the number of arguments. The backward mode is closely related to sensitivity analysis [9, 10], in which the function's derivative with respect to an intermediate variable is named as the "adjoint" of the variable. In each elementary operation, if the adjoint of the result and the values of the operands are known in advance, the adjoints of the operands can be calculated. So in the backward mode, all intermediate variables are evaluated first, and then the adjoints are calculated, proceeding from the final result to the arguments. When we give "work(g)" and work(g, ∇g)" more concrete definitions, i.e. assume that an addition is cheaper than a multiplication and a division takes at least 50 % more work than a multiplication, and also include memory fetches/stores as the cost, it can be proven [13] that for the backward mode, $\frac{\text{work}\{g,\nabla g\}}{\text{work}\{g\}}$ is bounded under 5, regardless of the number of the arguments.

In general, AD tools can be divided into two classes: the source code translator and the external library. Until today, numerous AD tools have been developed, and some representative tools are as follows: ADIFOR—a forward/backward Fortran 77 code translator [6]; OpenAD—a forward/backward mode XML schema translator [26, 27]; FADBAD—a forward/backward mode C++ library based on runtime polymorphism [5]; ADETL—a forward mode C++ library based on expression templates [33, 34]; Sacado—a forward/backward mode C++ library [21, 22], in which the forward mode is realized by expression templates; and ADEPT—a backward mode C++ library based on expression templates [15]. Source translators convert the original code which only evaluates variables to code that calculates corresponding gradients. This method is less desirable in reservoir simulators, because making changes to the program code would become less convenient. External libraries act as extensions to the current language to provide additional data structures and overloaded elementary operators on these data types. Each data structure packs one variable value and the associated derivatives into a capsulation. With external libraries, the AD code will have a similar appearance to the plain evaluating expressions, and the change in the expression is directly reflected in the gradients change. External libraries are preferred in modern reservoir simulators. Researchoriented simulators, such as MRST [18] and GPRS [11], as well as commercial simulators, such as Schlumberger Intersect [12], have already introduced AD libraries to simplify the coding of gradients. In a recent modification, GPRS uses ADETL to reconstruct most of its formulations and is, hence, renamed as AD-GPRS [33, 34], in which the data from basic independent variables to the residual vectors are all AD structures. The gradient part of the residual vector is used to construct the Jacobian matrix. The heavy use of AD makes AD-GPRS completely different from its predecessor.

The biggest obstacle to applying AD libraries to reservoir simulators lies in computational inefficiency. The forward mode based on operator overloading, which is the most user-friendly, may suffer from the issue of temporary objects. Upon the return of each elementary operation, the object that stores the result is delivered to a newly allocated temporary object as the returned value. The resultant object is then destroyed upon the exiting of the current function call, and the returned object is destroyed after the function call which takes it as an argument. As the length of the gradient vector is usually determined at runtime, the construction and destruction of these short-life temporary objects occur on the heap memory, which causes substantial performance deterioration of more than an order of magnitude [33]. The "expression templates" technique [1, 28] can overcome this issue. Expression templates are a kind of compile-time polymorphism (in comparison with runtime polymorphism). However, the work ratio of the forward mode is independent of polymorphism implementations. Sacado introduces caching and expression-level reverse mode [7] techniques to reduce the unnecessary calculations of forward mode. In fact, Sacado tries to combine the forward mode with the backward mode to some extent. The ADEPT has realized a template-based backward mode with the help of global stacks, and the benchmarks show that it has better performance than the runtime forward/backward AD and the template based forward AD in both speed and memory use.

The backward mode is superior to the forward mode when the expression is complicated. It naturally avoids constructions and copies of temporary objects and, meanwhile the work ratio is independent of the argument list length. Though the evaluation phase and the differentiation phase will both result in recursive function jumps, which take additional time, it is much cheaper than the heavy allocating/de-allocating operations on heap memory. In addition, with expression templates the recursive functions can be expanded during the compiling stage. In this work, we propose a backward mode AD implementation that is independent of global variables. As this scheme is essentially thread-safe and without assessing stacks, the expanded functions are less disruptive to CPU pipelines.

For the sake of simplicity, in the following text, we refer to the backward mode AD realized by runtime polymorphism as the “runtime backward AD” and refer to the backward mode AD realized by expression template as the “compile-time backward AD” The remainder of this paper is organized as follows: in Section 2, we demonstrate how to implement backward mode with runtime polymorphism or with expression templates, and provide a method to perform variable substitution automatically and rapidly; and in Section 3, we discuss the feasibility of applying this “backward AD + variable substitution” procedure in a reservoir simulator and compare the performance of the runtime backward AD, the compile-time backward AD, and the HD. The programming language in our discussion is assumed to be C++0x; the compiler that we use is Visual C++10.

2 Methods

2.1 The backward mode AD

2.1.1 Basic concepts of AD

The basic principle of AD is the chain rule, namely if $\frac{dh}{dx}$ is known, $\frac{d}{dx}\left[g\left(h\right)\right]$ equals $\frac{dg}{dh}\frac{dh}{dx}$, where x, h, and g may be vectors. Usually the form of $g\left(h\right)$ is not simple and can be decomposed into a sequence of elementary operations: $g\left(h\right) = g_K\left(g_{K-1} \ldots \left(g_1\left(h\right)\right)\right)$, where $g_i\ (i = 1 \cdots K)$ stands for binary operations such as multiplications and additions, or unary operations such as logarithms and exponentiations. Each elementary operation has a particular deviation rule, e.g., $(h_1 \times h_2)' = h_1' h_2 + h_2' h_1$, $[\ln(h_1)]' = h_1'/h_1'$. The goal of AD is to create code that automatically propagates derivatives from $\frac{dh}{dx}$ to $\frac{dg}{dx}$. The intermediate propagating process can be performed either from $\frac{dg_1}{dh}$ to $\frac{dg}{dh}$ (the forward mode), or from $\frac{dg}{dg_K}$ to $\frac{dg}{dh}$ (the backward mode). When propagating from $\frac{dg}{dg_K}$ to $\frac{dg}{dh}$, the values of all intermediate variables $(g_K, g_{K-1}, \cdots g_1)$ should be known in advance; so, backward AD needs a standalone evaluating phase at the beginning.

In the backward AD, the specially defined data structure stores the variable’s value and the adjoint, which is the final result’s derivative with respect to this variable. The backward AD converts an expression to a directed acyclic graph (DAG), which is also referred to as the “parse tree” The parse tree can be reused if the expression is evaluated multiple times. Figure 1 shows the parse tree of Eq. 1 as an example. The leaf nodes stand for the independent variables, and the root node stands for the final result.

$$q_\mathrm{P} = (\lambda_\mathrm{P} \times T_{12}) \times (p_2 - p_1 - \Phi_H) \qquad (1)$$

In reservoir simulations, Eq. 1 is the general form of phase P’s flow rate from grid block 2 to grid block 1, where λ_P is the fluid mobility; T_{12} is the transmissibility; p_1 and p_2 are block center pressures; and Φ_H is the gravity potential difference. The backward AD calculates all intermediate variable values and all adjoints in separate phases. First, the intermediate variable values are evaluated from the leaves to the root; next, the adjoints are evaluated from the root to the leaves. In Fig. 1, if we obey the right side priority, the values are evaluated following the sequence “$V_1 \to V_2 \to V_3 \to V_4$” (Fig. 1 left), while the adjoints are evaluated following the sequence “$V_4 \to V_2 \to V_1 \to V_3$” (Fig. 1 right). The adjoint of the root node always equals 1. By starting from this, all subordinate adjoints can be evaluated according to the chain rule, e.g., $\frac{\partial q_\mathrm{P}}{\partial V_2} = \frac{\partial q_\mathrm{P}}{\partial V_4}\frac{\partial V_4}{\partial V_2} = 1 \cdot V_3$. Compared with the backward mode, the forward mode evaluates the derivatives together with the values; so, it has only one phase.

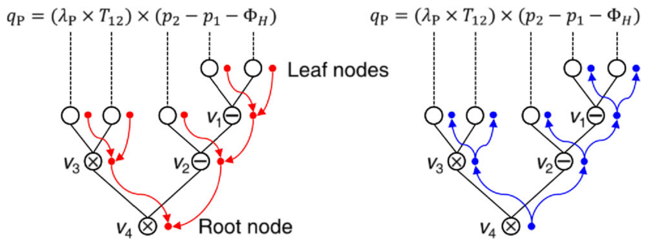
$q_\mathrm{P} = (\lambda_\mathrm{P} \times T_{12}) \times (p_2 - p_1 - \Phi_H)$
Leaf nodes
V_1
V_2
V_3
V_4
Root node
$q_\mathrm{P} = (\lambda_\mathrm{P} \times T_{12}) \times (p_2 - p_1 - \Phi_H)$
V_1
V_2
V_3
V_4

Fig. 1 An example of a parse tree in backward AD. *Leaf nodes* (the *empty circles*) represent the arguments of the expression. The circles with operators in them represent the results of elementary operations. The evaluation sequence follows the *red arrows*; the differentiation sequence follows the *blue arrows*

2.1.2 Backward mode by runtime polymorphism

It is natural to implement the backward AD with runtime polymorphism, which is the case in most currently available backward AD libraries. For simplicity, we call this the "runtime backward AD". In this approach, the leaf node, the constant node and the intermediate nodes that carry elementary operations are all derived from a base type. The base type only stores the value and the adjoint, and it will never be involved in calculation. The derived types are different from each other: the leaf node also stores its index; the constant node has an adjoint always being zero; and the intermediate node stores the references of its one or two operands additionally. The building of the parse tree through overloaded operators is simply to pass the references of the two (or one) operands to its parent node corresponding to the operator. Then we can traverse the whole parse tree by recursively querying the children of the top parent node (the root node). Another difference between the derived nodes lies in their evaluating and differentiating routines. The base type only provides pure virtual definitions of these functions, while each derived type provides the concrete realizations. For example, in the multiplication node—the "MulNode" class, the evaluating function is as follows:

```
void MulNode::Eval (const double* Varg)

{ Right->Eval (Varg); Left->Eval (Varg);

   value = Left->value * Right->value; }
```

the differentiating function is as follows:

```
void MulNode::Diff(double* Grad)

{ Right->adjoint = adjoint * Left->value; Right->Diff(Grad);

  Left->adjoint = adjoint * Right->value; Left->Diff(Grad); }
```

where "Varg" is an array that stores the values of the expression's arguments; "Grad" is an array that stores the output gradients; and "Left" and "Right" are references to the left and right operands, respectively. "Left" and "Right" are base type pointers, which have the authority to access any derived types and invoke their data and functions inherited from the base type.

Calling the "EVal" function and the "Diff" function of the root node will form recursive callings of all subordinate "EVal" functions and all subordinate "Diff" functions, respectively. The recursions terminate at the leaf nodes or the constant nodes. The constant node has completely empty evaluating function and differentiating function; whereas, the leaf node has the evaluating function as follows:

```
void Leaf::Eval(const double* Varg) { value = Varg[IDX]; }
```

and the differentiating function as:

```
void Leaf::Diff(double* Grad) { Grad[IDX] += adjoint; }
```

where "IDX" is the index of a leaf node. So in the leaf node, the "Eval" function is simply to get the node value from the input array, and the "Diff" function is simply to accumulate the gradient to the output array.

During the evaluation or the differentiation phase, the backward mode AD need not return temporary objects. This is a significant advantage of the backward mode AD over the forward mode AD. It may be argued that the construction of the parse tree takes additional time. However, in reservoir simulations an expression is usually used hundreds of thousands of times; thus, the construction time can be ignored. The real problem is the expensive recursive function jump in/out. Under certain circumstances the compiler can automatically expand recursive functions. Unfortunately, however, here the compiler will reject expanding them, because "Eval" and "Diff" are virtual functions to be determined dynamically and which concrete realization of "Eval" or "Diff" will be used is not known to the compiler.

2.1.3 Backward mode by expression templates

Expression templates are a kind of metaprogramming paradigm, which attempts to utilize compilers to make all possible choices and computations at compile time to generate fast programs [1]. Expression templates are first described by Veldhuizen [28] and have led to the development of the high performance vector arithmetic library Blitz++ [29]. The first expression-template-based AD dates back to 2001 when Aubert et al. [2] applied it in a flow control problem. In recent years, expression-template-based AD has been introduced to reservoir simulators [12, 33–35]. However, these AD implementations all make use of the forward mode which is convenient for application but computationally inefficient for sophisticated expressions. The goal of this work is to address the most complicated expressions in reservoir simulators with AD. Under this premise, the backward mode may be the most appropriate. The recently published ADEPT [15] implemented the expression-template-based backward mode AD, but it relies on global stacks that record indices and adjoints, making the library less convenient for reservoir simulators, especially when parallelization is considered.

We propose an expression-template-based backward mode AD that is independent of global variables. For simplicity, we call this realization of AD the "compile-time backward AD”. The basic idea is to let the intermediate nodes carry adjoints themselves and enable compiler’s optimization to the recursive calls of evaluating and differentiating function. With C++ templates, runtime polymorphism can be replaced by compile-time polymorphism. In the compile-time backward AD, the evaluating and differentiating functions of the leaf node, the intermediate node and the constant node still keep similar forms as they are in the runtime version. However, the type of references to the left and right operands changes. Now they are abstract references of any type, and the type declarations themselves become the template arguments of the node. For example, the compile-time version of multiplication node is defined as follows (the contents of other functions are omitted here except for the construction function):

```
    template<typename LeftType, typename RightType>
    class MulNode<LeftType, RightType> {
    public:
    MulNode(LeftType& _Left, RightType& _Right)
    { Left = _Left; Right = _Right; };
    void Eval(const double* Varg);
    void Diff(double* Grad);
    double value, adjoint;
    LeftType&    Left;
    RightType& Right;
};
```

“MulNode” is one of the intermediate nodes. It cannot be declared explicitly but only be specialized upon the calling of the multiplication function, which is defined using the operator overloading technique:

```
template<typename LeftType, typename RightType>
MulNode<LeftType, RightType> operator*(LeftType& left, RightType& right)
{ return MulNode<LeftType, RightType>(left, right); };
```

Both operands should be either a constant node, a leaf node, or an intermediate node. Consider an expression “a*(b*c)” built by expression templates, where a, b, and c are leaf nodes. The expression will have a nested type expressed as “MulNode< Leaf, MulNode< Leaf, Leaf>>”. During the construction of the expression, the evaluating and differentiating function will be expanded by the compiler, because the sequence of the subordinate function calls are completely static. The expanded “Diff” function of “a*(b*c)” is equivalent to the following:

```
Grad[c.IDX] = Grad[c.IDX] + a.value*b.value;
Grad[b.IDX] = Grad[b.IDX] + a.value*c.value;
Grad[a.IDX] = Grad[a.IDX] + b.value*c.value;
```

Mainstream C++ compilers (e.g., Visual C++, g++, Intel C++) support deep expansion to inline functions.

2.1.4 Comparison of runtime and compile-time backward AD

In the runtime backward AD, the parse tree is built dynamically and stored on heap memory. All child nodes can be accessed through the root node, i.e., the root node owns the tree. Sometimes root node A is not only used by one expression, but can also be one branch under another root node B. Upon the deletion of B, destruction is executed recursively on its child nodes. To avoid the unexpected destruction of A before the termination of A’s life cycle, each intermediate node should count how many times it has been referred to, i.e., count how many red arrows that it launches as shown in Fig. 1. When destruction is executed to the node, the counter decreases by 1 and, when the counter equals zero, the delete-instruction is sent down to its child nodes. With this reference counting technique, it is safe to deliver a dynamic parse tree to references, and then reuse it as branches of other root nodes.

The runtime backward AD naturally allows conditional branches while the compile-time backward AD is less flexible. With all expressions being determined statically, the conditional branches are not allowed. The whole expression should be rewritten in different conditional branches. Additionally, the templated root node always has a deep nested type deduced by the compiler and, thus, is hard to deliver to a reference, because the type of reference is not intuitive. To reuse the templated root nodes as branches can be only achieved by macro definitions. In the compile-time backward AD, even the calling of the evaluating or differentiating function is not direct. We should create a templated interface that accepts abstract root node types:

```
template<typename RootType>
double RADEval(RootType& Root, const double* Varg, double* Grad)
{ Root.Eval(Varg); Root.Diff(Grad); return Root.value; };
```

The example code for evaluating and differentiating “a*(b*c)” with expression templates is given in Appendix 1. In addition, a more practical example is given in Appendix 2.

Despite these inconveniences, as we will demonstrate in Section 3.2, the compile-time backward AD is more computationally efficient. So we suggest using compile-time backward AD whenever possible and only using runtime backward AD in conditional expressions that are not frequently executed, such as the well in/out flow rate. In other cases, the expression should be dissembled, with the static parts realized by the compile-time backward AD.

2.2 The automatic variable substitution

Our ultimate goal of differentiating the complicated expressions is to evaluate the Jacobian matrix. With either the runtime or the compile-time backward AD, the gradients of an expression with respect to the arguments are available. In most cases, however, the arguments are not the independent variables of the reservoir simulation equations. Substituting the arguments with the independent variables for a function f results in the matrix-vector multiplication, which can be written in general as:

$$\mathbf{f}_x = \sum_{i=1}^{N} \left(\frac{\partial f}{\partial a_i} \mathbf{a}_{i,x} \right) \tag{2}$$

where

$(x_1, x_2, \cdots, x_M)$ is the basic variable set;
$(a_1, a_2, \cdots, a_N)$ is the argument set of f;
$\frac{\partial f}{\partial a_i}$ $(i = 1, \cdots, N)$ is already calculated by backward mode AD;
$\mathbf{a}_{i,x} = \left(\frac{\partial a_i}{\partial x_1}, \frac{\partial a_i}{\partial x_2}, \cdots, \frac{\partial a_i}{\partial x_M} \right)$, imported from other modules;
$\mathbf{f}_x = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_M} \right)$, $\mathbf{f}_x$ is the ultimate gradient vector that we need to evaluate the Jacobian matrix

Variable substitutions occur frequently in reservoir simulations. Taking Eq. 1 as an example, under the natural variable set, which is widely accepted in compositional models, none of the arguments λ_P, T_{12}, and Φ_H belong to the basic variable set. In the most popular object-oriented development of a reservoir simulator, useful intermediate variables, such as transmissibilities, fluid motilities, gravity potential differences and phase pressures, all with their ultimate gradient vectors, are provided by individual modules. The flow calculation module only takes the responsibility for evaluating the flow rate and assembling its derivatives.

Usually a dependent variable is only related to a small portion of the independent variables. To save space, the gradient vector is stored sparsely, i.e. only nonzero entries and their indices are stored. If $\mathbf{a}_{i,x}$ is sparse and $\mathbf{f}_x$ is dense (Fig. 2 I), the entry of $\mathbf{a}_{i,x}$ with index equal to k is directly mapped to the kth entry of $\mathbf{f}_x$, which is equivalent to sparse matrix to dense vector multiplication. If both $\mathbf{a}_{i,x}$ and $\mathbf{f}_x$ are dense (Fig. 2 II), no index aligning is required and the floating point operations may be accelerated by singe-instruction-multiple-data (SIMD) instructions, hence being effective if the vectors are nearly full. The situation in which $\mathbf{a}_{i,x}$ is dense and $\mathbf{f}_x$ is sparse rarely occurs in reservoir simulations (Fig. 2 III). In the following, we will not discuss the first two situations, for there are numerous BLAS libraries to handle them, but only focus on the situation in which both $\mathbf{a}_{i,x}$ and $\mathbf{f}_x$ are sparse (Fig. 2 IV), which is called the point sparse mode. However, this does not mean that we only utilize this mode in reservoir simulations. In fact, we use point sparse mode mainly in simple property calculations and, for properties that are more complex, such as the flux stencil, we use the block sparse mode in which the vector operations are dense to dense in each grid block.

Manual coding for the "sparse to sparse" combination is tedious because the sparse pattern of $\mathbf{f}_x$ depends on every sparse pattern of $\mathbf{a}_{i,x}$ $(i = 1, 2, \ldots)$. The code should be modified if any $\mathbf{a}_{i,x}$ $(i = 1, 2, \ldots)$ changes its sparse pattern, and a catastrophic change in code will occur if the basic unknown set changes. However, the sparse patterns can still be merged automatically. Consider two index arrays if their first entries are not equal, pop-front the smaller one and push-back that entry to a new queue, and if their first entries are equal, pop-front both entries and push-back only one of them; repeat this operation until both arrays are empty, then the new queue can be merged with the next index array. However, this routine would waste too much time if we

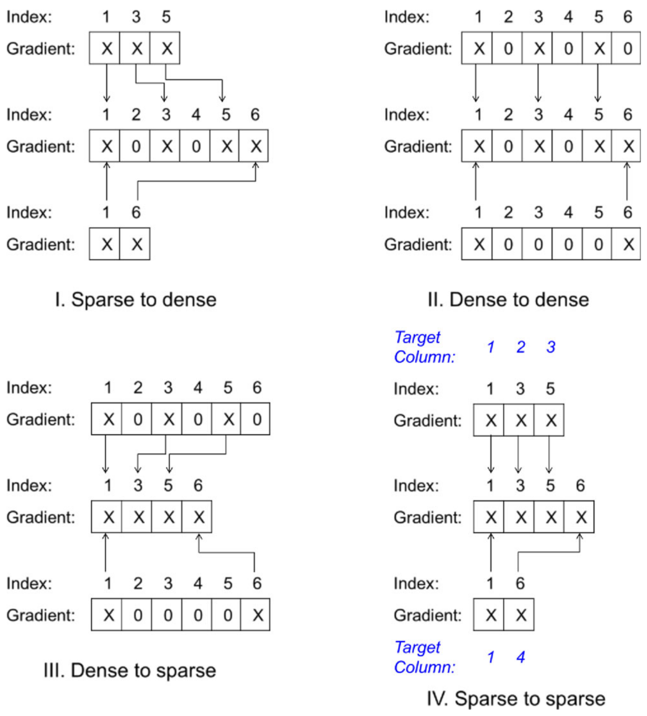

Fig. 2 Four situations of merging gradient vectors. The "sparse to sparse" situation is accelerated by recording the target column numbers

merge the same set of index arrays for tens of thousands of times because, from the second time the comparing of indexes and the allocating of arrays are unnecessary. After the first time of merging, the sparse pattern of the $\mathbf{f}_x$ is known. We can compare $\mathbf{a}_{i,x}$ with $\mathbf{f}_x$ to determine which column of $\mathbf{f}_x$ has the same index as the first entry of $\mathbf{a}_{i,x}$. We record that number, and the next time when we accumulate $\frac{\partial f}{\partial a_i}\mathbf{a}_{i,x}$ to $\mathbf{f}_x$, the first entry of $\mathbf{a}_{i,x}$ is accumulated directly to that column of $\mathbf{f}_x$. The same work is done for the rest of the entries of $\mathbf{a}_{i,x}$; so, the next time when combining sparse vectors, each entry of $\mathbf{a}_{i,x}$ knows into which column of $\mathbf{f}_x$ it should go. This procedure is demonstrated in the lower right part of Fig. 2. By recording target column numbers, we shift the "sparse to sparse" merging to the "sparse to dense" merging, and the latter is free of indices aligning.

This "target column number recording" raises three questions. The first is whether it depletes the memory. Fortunately in reservoir simulations, all models have limited mathematical forms, i.e., all expressions in the equations have limited lengths and limited conditional branches, and the stenciling operations only involve limited grid blocks (usually two). Hence, the amount of cached column numbers is limited. The second question is where we should store the target column numbers. In an object-oriented simulator, each module can keep its own state data, so the target column numbers can be stored as a part of the module's state. During the initialization of a module, the sparse pattern of the output gradient vector is determined according to the sparse patterns of all input gradient vectors; then, the target column numbers are determined. The third question is how to handle independent variable change which occurs when phase appears or disappears. In our framework, the selecting of the primary variables is carried out in the solution stage, and the secondary variables are eliminated using Schur complement (Fig. 3). When oil and gas co-exist, the Schur complement is phase equilibrium constrain; otherwise it is a trivial matrix. In the linearization stage, the governing equations are differentiated regardless of phase number—derivatives with respect to all independent variables are calculated, so the variable substitution code is completely static.

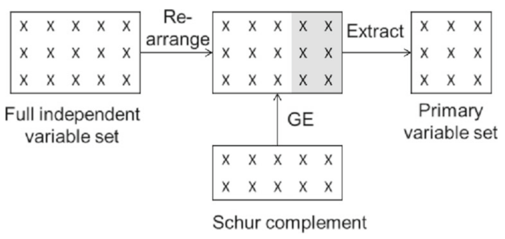

Fig. 3 Reducing unknown variable dimension with Gaussian elimination (GE). The matrix columns are re-arranged, and derivatives with respect to secondary variables (the *gray part*) are eliminated. Here, only one block of the Jacobian matrix is illustrated, corresponding to one reservoir grid

3 Applications

3.1 The feasibility of the backward AD framework

Our AD framework can eliminate the heavy manual differentiation work and let the developers focus on the model physics and other aspects. It is also noted that our framework has good efficiency because (1) the backward AD has lower computational costs than the forward AD in expressions with long argument lists; (2) the expression templates further eliminate the recursive function calling; and (3) by recording target column numbers, the repeated "sparse to sparse" vector combination has equal speed with the "sparse to dense" vector combination.

The framework is naturally compatible with object-oriented simulator development, in which the "least knowledge principle" [19], i.e., modules should hide details from each other, is obeyed. In our framework, if a module that computes a certain kind of quantity is initialized, the sparse pattern of the output gradient vector and the cached target column numbers depend only on the sparse pattern of input gradient vectors and the mathematical model contained in

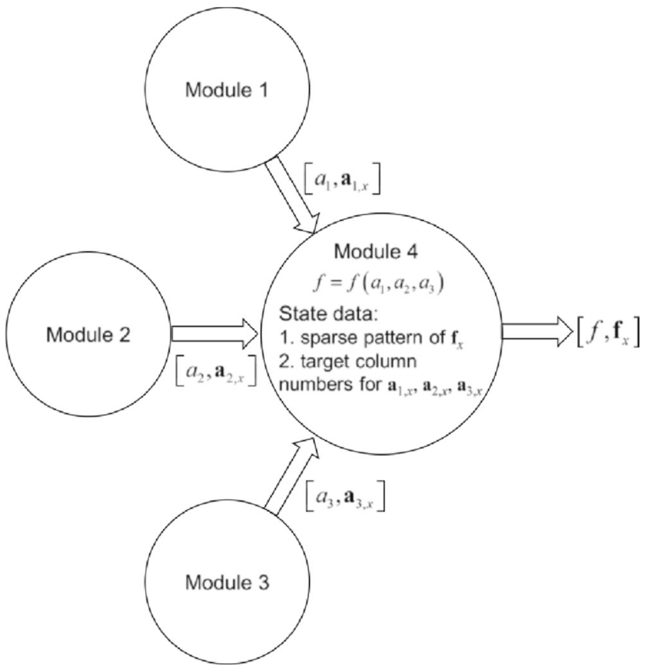
Module 1
Module 2
Module 3
Module 4
$f = f(a_1, a_2, a_3)$
State data:
1. sparse pattern of $\mathbf{f}_x$
2. target column numbers for $\mathbf{a}_{1,x}$, $\mathbf{a}_{2,x}$, $\mathbf{a}_{3,x}$
$[a_1, \mathbf{a}_{1,x}]$
$[a_2, \mathbf{a}_{2,x}]$
$[a_3, \mathbf{a}_{3,x}]$
$[f, \mathbf{f}_x]$

Fig. 4 The present AD framework hides module details. For example, Module 4's state data rely only on the sparse pattern of $\mathbf{a}_{i,x}$ $(i = 1, 2, 3)$, and the mathematic forms inside of Module 4. Other details in Module 1, 2, or 3 are not required

the module and do not include the details of how these input patterns are produced. The independence of the module is kept as much as possible (Fig. 4).

3.2 Individual tests on some expressions

Case 1 to case 6 are tests on individual expressions, for which we use different differentiation approaches and compare the performances. All cases in this paper, including the cases in Section 3.3 are compiled by Visual C++ 10 and run on a 2.8 GHz Core 2 CPU in one core mode.

3.2.1 Validation of the compiler optimization

Case 1 $y_N = \prod_{i=1}^{N} \left(\sum_{j=1}^{i} x_j \right)$

The first issue to confirm is that the inline expansion is effective to the compile-time backward AD. We use both runtime and compile-time backward AD coding for case 1, and compile it under different combinations of compiler optimization options, one being "non-optimized" (the compiler optimization option is "/Od/Ob2") and the other being "fully optimized" (the compiler optimization option is "/O2/Ob2/Oi/GL"). If the compiler expands the nested functions of the backward AD, the "fully optimized" version would have a great improvement in speed, because the expensive function jumps are avoided The GFLOPS (10^9 floating point operations (FLOPs) per second, i.e., GFLOPS $= \frac{n\,(\text{FLOPs})/10^9}{\text{time(s)}}$) of repeatedly running case 1 (with $N = 5, 10, 15, 20, 25$, respectively) are illustrated in Fig. 5. A comparison with the ADEPT is also made in this case. The ADEPT is compiled under the "fully optimized" option, and the thread unsafe stack is used.

In our approach, the non-optimized runtime backward AD and the non-optimized compile-time backward AD have nearly the same GFLOPS, but after the compiler's optimization, the compile-time version is 2.6 to 2.9 times faster than the runtime version, indicating that the inline expansion has taken effect. We have further confirmed this by checking the compiler-optimized assembly code and found that most of the intermediate nodes do not actually exist, which means that the nested expressions are executed similar to hand written code. Compared to the ADEPT, our approach has better performance. The "fully optimized" compile-time backward AD is 1.5 to 2.3 times faster than the "fully optimized" ADEPT. This is because accessing stacks makes the execution of code more interruptive to CPU pipelines. In fact the "fully optimized" ADEPT is close to our "fully optimized" runtime version. However, it is true that some flexibility would be lost if stacks are not used. The self-operators (e.g., $+=$, $*=$) are not available in our compile-time backward AD.

In our approach, the "fully optimized" compile-time backward AD can reach a speed of 0.2 to 0.3 GFLOPS. The peak GFLOPS of this CPU in one core mode is 3.43, which is obtained through testing dense matrix-to-matrix multiply subroutine (DGEMM) of Intel Math Kernel Library (MKL). However, this is an overly optimistic value because MKL's DGEMM benefits from the SIMD instructions but this is not the case in backward AD. In reservoir simulation programs, as the logic is usually complicated, it is also very difficult to widely apply the SIMD instructions

Case 1 also shows the advantage of the backward mode over the forward mode. Define "*work*$(\cdot)$" to be the number of FLOPs. With backward mode AD, $\frac{\text{work}(y_N, \nabla y_N)}{\text{work}(y_N)} \approx 3.0$; while with forward mode AD, $\frac{\text{work}(y_N, \nabla y_N)}{\text{work}(y_N)} \approx 1 + \frac{N}{3}$

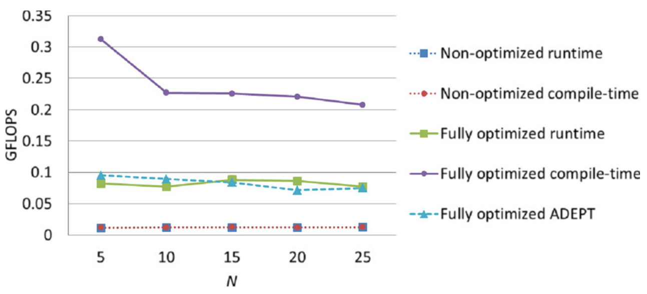

Fig. 5 Performance comparison of different AD implementations under different compiler optimization options. The results of ADEPT are also included

3.2.2 Tests on complicated expressions in reservoir simulation equations

The proposed framework is implemented in an in-house reservoir simulator—Unconventional Oil and Gas Simulator (UNCONG) as an external library. UNCONG aims to model the most up-to-date gas and oil recovery processes. It is a highly modularized, easy-to-extend simulator, with the capability of modeling industrial scale black oil/compositional problems. Some models in UNCONG have complicated mathematic forms. Since the derivatives are difficult to obtain and code by hand, they are realized by this AD framework. We choose some of them to perform benchmark tests (case 2 to case 6). These cases are described as follows:

Case 2 The matrix density of the fractured coal in the dual porosity CBM model [30]

$$\rho_s = (1 - \phi_f - \phi_m) \cdot \left[\rho_{c0} + C_s \rho_{c0} \left(\frac{p_f \phi_f + p_m \phi_m}{\phi_f + \phi_m} - p_0 \right) \right] \quad (3)$$

The arguments are as follows: ϕ_f—the fracture porosity; ϕ_m—the matrix porosity; p_f—the fracture pressure;

and p_{m}—the matrix pressure. The constants are as follows: p_{c0}—the reference coal density; C_{s}—the compressibility of coal; and $p_0$the reference pressure. The basic variable set is $\{p_{\mathrm{wf}}, S_{\mathrm{wf}}, p_{\mathrm{wm}}, S_{\mathrm{wm}}\}$, where p_{wf} is the fracture water pressure; p_{wm} is the matrix water pressure; S_{wf} is the fracture water saturation; and S_{wm} is the matrix water saturation. ϕ_{f} and p_{f} are functions of $(p_{\mathrm{wf}}, S_{\mathrm{wf}})$; and ϕ_{m} and p_{m} are functions of $(p_{\mathrm{wm}}, S_{\mathrm{wm}})$.

Case 3 The Forchheimer non-Darcy flow [16] for gas

The volumetric flow rate q_{g}satisfies the following:

$$\nabla\Phi_{\mathrm{g}} = \frac{1}{C_1}\left(\frac{\mu_{\mathrm{g}}}{K k_{\mathrm{rg}} A}\right) q_{\mathrm{g}} + \frac{C_2}{C_1}\beta\rho_{\mathrm{g}}\left(\frac{q_{\mathrm{g}}}{A}\right)^2 \tag{4}$$

q_{g} is then expressed as the positive root of this quadratic equation. The arguments of q_{g}'s expression are as follows: $\nabla\Phi_{\mathrm{g}}$—potential gradient of gas phase; μ_{g}—gas viscosity; k_{rg}—relative permeability of gas; and ρ_{g}—gas density. The constants are C_1, C_2, K, A and β. The independent variable set is $\{p_{\mathrm{o}}, S_{\mathrm{w}}, S_{\mathrm{g}}, R_{\mathrm{s}}\}$ (black oil model). The dependences between the arguments and the basic variable set also follow the black oil model. The constants are merged before calculation.

Case 4 Tubing flow friction force in the multi-segment well (MSWell) model [17]

$$p_{\mathrm{f}} = \bar{\rho}\left(\frac{2C_{\mathrm{f}} f V_{\mathrm{m}} |V_{\mathrm{m}}|}{D}\right) L \tag{5}$$

where

$\bar{\rho}$	is the average fluid density, weighted by phase volume fractions;
C_{f}	is a unit converting factor;
f	is the Fanning friction factor;
V_{m}	is the average velocity;
D	is well diameter;
L	is well segment length.

The Fanning friction factor—f is calculated using Haaland's correlation [14]:

$$f^{-0.5} = -3.6\lg\left[\frac{6.9}{Re} + \left(\frac{\varepsilon}{3.7D}\right)^{1.11}\right] \tag{6}$$

where

ε is the roughness factor of the tube wall;

Re is the Reynolds number: $Re = C_{\mathrm{r}}\frac{\bar{\rho}V_{\mathrm{m}}D}{\bar{\mu}}$, where C_{r} is a unit converting factor; and $\bar{\mu}$ is the average fluid viscosity, weighted by phase volume fractions.

The arguments are: $\bar{\rho}$, $\bar{\mu}$, and V_{m}; the constants are C_f, D, L, $\varepsilon C_{\mathrm{r}}$. The independent variable set is $\{p, V_{\mathrm{m}}, \alpha_{\mathrm{g}}, \alpha_{\mathrm{w}}, R_{\mathrm{s}}\}$ (black oil MSWell model). The dependence between the arguments and the basic variable set also follows the black oil MSWell model. The constants are merged before calculation.

Case 5 Gas drift velocity for liquid + gas flow in the MSWell model [24]

$$V_{\mathrm{gd}} = \frac{A \cdot B \cdot (K_{\mathrm{u}})_i \cdot (V_{\mathrm{c}})_{i-1} \cdot \sqrt{(\rho_{\mathrm{l}})_{i-1}}}{\sqrt{(\rho_{\mathrm{g}})_i} \cdot B + \sqrt{(\rho_{\mathrm{l}})_{i-1}} \cdot A} \tag{7}$$

where

Subscript "$i-1$" denotes variable in the "downstream" well segment, the segment to which the fluid in segment i flows;

$A = (1-\alpha_g C_0)_{i-1}$, where α_{g} is gas holdup and C_0 is a profile parameter;

$B = (\alpha_{\mathrm{g}} C_0)_i$

K_{u}	is the critical Kutateladze number;
V_{c}	is the characteristic velocity;
ρ_{g}	is the gas density;
ρ_l	is the average liquid density: $\rho_l = (\alpha_{\mathrm{o}}\rho_{\mathrm{o}} + \alpha_{\mathrm{w}}\rho_{\mathrm{w}}) / (\alpha_{\mathrm{o}} + \alpha_{\mathrm{w}}\rho_{\mathrm{w}})\,(\alpha_{\mathrm{o}} + \alpha_{\mathrm{w}})$; where α_{w} is water holdup; α_{o} is oil holdup ($\alpha_{\mathrm{o}} = 1 - \alpha_{\mathrm{w}} - \alpha_{\mathrm{g}}$), and ρ_{o}, ρ_{w} are oil and water density, respectively.

In the black oil MSWell model, the independent variable set is $\{p, V_{\mathrm{m}}, \alpha_{\mathrm{g}}, \alpha_{\mathrm{w}}, R_{\mathrm{s}}\}$. ρ_o is a function of (p, R_{s}); ρ_{g} and ρ_{w} are both functions of p; K_{u} and V_{c} are both functions of $(p, \alpha_{\mathrm{g}}, \alpha_{\mathrm{w}}, R_{\mathrm{s}})$; and C_0 is a function of $(p, V_{\mathrm{m}}, \alpha_{\mathrm{g}}, \alpha_{\mathrm{w}}, R_{\mathrm{s}})$. Assuming that ρ_{o}, ρ_{g}, ρ_{w} α_{g} K_{u} V_{c} and C_0, as well as their gradient vector with respect to the basic variable set, have already been provided by other modules.

Case 6 Peaceman's formula [20] to calculate well index (WI)

$$\begin{aligned}
\mathrm{WI}_x &= \mathrm{h_x}\sqrt{k_y k_z}\Bigg/\left[\ln\left(\frac{0.28}{\mathrm{r_w}}\frac{\sqrt{k_y \mathrm{D_z^2} + k_z \mathrm{D_y^2}}}{\sqrt{k_y}+\sqrt{k_z}}\right) + \mathrm{s}\right]\\
\mathrm{WI}_y &= \mathrm{h_y}\sqrt{k_z k_x}\Bigg/\left[\ln\left(\frac{0.28}{\mathrm{r_w}}\frac{\sqrt{k_x \mathrm{D_z^2} + k_z \mathrm{D_x^2}}}{\sqrt{k_x}+\sqrt{k_z}}\right) + \mathrm{s}\right]\\
\mathrm{WI_z} &= \mathrm{h_z}\sqrt{k_x k_y}\Bigg/\left[\ln\left(\frac{0.28}{\mathrm{r_w}}\frac{\sqrt{k_y \mathrm{D_x^2} + k_x \mathrm{D_y^2}}}{\sqrt{k_x}+\sqrt{k_y}}\right) + \mathrm{s}\right]\\
\mathrm{WI} &= \left(\mathrm{WI}_x^2 + \mathrm{WI}_y^2 + \mathrm{WI}_z^2\right)^{1/2}
\end{aligned} \tag{8}$$

where k_x, k_y and k_z are arguments; $\mathrm{D_x}, \mathrm{D_y}, \mathrm{D_z}$, $\mathrm{h_x}$, $\mathrm{h_y}$, $\mathrm{h_z}$, $\mathrm{r_w}$ and s are constants; and k_x, k_y and k_z are variable permeabilities related to the reservoir independent variable set—$\{p_{\mathrm{o}}, S_{\mathrm{w}}, S_{\mathrm{g}}, R_{\mathrm{s}}\}$. The constants are merged before calculation.

Table 1 The work ratios and FLOP numbers of backward mode AD and the forward mode AD, the FLOP numbers of the expression differentiation stage and of the variable substitution stage of the backward AD are recorded separately in quotes

	Backward AD work ratio	Forward AD work ratio	Backward AD FLOPs (expression differentiation + variable substitution)	Forward AD FLOPs
Case 2	2.58	5.16	47 (31 + 16)	94
Case 3	2.00	4.60	60 (42 + 18)	138
Case 4	2.70	9.62	64 (46 + 18)	228
Case 5	2.96	10.4	128 (80 + 48)	450
Case 6	2.74	6.78	140 (116 + 24)	346

Table 2 The time cost (ms) of different realizations of case 2 to case 6, the time cost of the expression differentiation stage and of the variable substitution stage of the backward AD are recorded separately in quotes

	Backward AD		Forward AD	HD
	Runtime version (runtime expression differentiation + variable substitution)	Compile-time version (compile-time expression differentiation + variable substitution)		
Case 2	296 (265 + 31)	132 (101 + 31)	427	74
Case 3	471 (429 + 42)	198 (156 + 42)	595	92
Case 4	695 (657 + 38)	242 (204 + 38)	1213	121
Case 5	938 (842 + 96)	408 (312 + 96)	2238	216
Case 6	1565 (1526 + 39)	599 (560 + 39)	1989	290

Table 3 GFLOPS of different realizations of case 2 to case 6

	Backward AD			Forward AD	HD
	Expression differentiation stage		Variable substitution stage		
	Runtime	Compile-time			
Case 2	0.117	0.307	0.516	0.220	0.635
Case 3	0.098	0.269	0.425	0.232	0.652
Case 4	0.070	0.226	0.469	0.188	0.529
Case 5	0.095	0.256	0.500	0.201	0.593
Case 6	0.076	0.207	0.615	0.174	0.483

Table 4 Description of reservoir simulation problems

	Model type	Grid size	Well block number
Case 7	Gas water	25,000	50
Case 8	Gas water	260,985	159
Case 9	Black oil	43,644	270
Case 10	Black oil	368,326	10
Case 11	Compositional (9 components)	324	2
Case 12	Compositional (9 components)	8748	6

For each case, we use the runtime backward AD, the compile-time backward AD the forward AD, and the HD, respectively, to evaluate the expression and its derivatives with respect to the arguments. In the backward AD and the HD versions, the gradients are transformed from being with respect to the arguments to being with respect to the basic variables after the expressions are differentiated with the target column recording technique. The HD code completely mimics the derivation process of backward mode AD; thus, its FLOP number equals that of the backward AD, which may not be optimal but serves as a reference. In forward AD versions, if the input argument is a dependent variable, we store its derivatives with respect to the basic variables as a dense vector before the calculation begins, and the derivatives with respect to the basic variables are calculated automatically when variables are merged, so the variable transformation stage is not needed. The forward AD is basically implemented by FADBAD++ v2.1 using the heap-based mode to allow the gradient vectors being of variable lengths; however we modified part of the source code to minimize the expensive allocating/de-allocating of temporary objects on heap memory when an operator function is returned, using the "right-value reference" technique, which is supported by C++0x. The work ratios (defined as in Section 3.2.1) and the FLOP numbers of the backward AD and of the forward AD are recorded in Table 1. All FLOP numbers are counted rigorously, with no distinctions between elementary operations and transcendental operations being made. Each case of each version is compiled under the "fully optimized" compiler option and run 10^6 times. The time costs of the four approaches for each case are recorded in Table 2; the GFLOPS of these approaches for each case are recorded in Table 3. Case 2 to case 6 are already sorted by the FLOP number. From the results, we can conclude the following:

1. The backward mode AD has a much lower work ratio than the forward mode AD.
2. The compile-time backward AD is 3.0 to 5.0 times the speed of the forward AD, and 2.2 to 2.9 times the speed of the runtime backward AD but only about 0.5 times the speed of HD.
3. The variable substitution takes a considerable fraction of time, and if the length of the independent variable set grows, the time cost by variable substitution will increase, hence lessening the overall performance deterioration caused by automatic differentiation of expressions.
4. In the backward AD, the GFLOPS (of either the expression differentiation stage or the variable substitution stage) is not affected much by the number of arithmetic operators or the number of arguments. Transcendental operators would affect GFLOPS (case 4 and case 6) because evaluating a transcendental function takes more CPU clock cycles. This situation is also true in the forward AD and the HD.

3.3 Tests in complete simulation processes

The final question is how much the proposed AD framework slows down the simulator. We answer this by comparing the time cost of the linearization work realized by different methods in complete reservoir simulations.

Cases 7 to 12 are reservoir simulation problems of different types and sizes. Detailed information on these cases is given in Table 4. We adjusted (P, T) of the compositional problems to make the fluid far from critical points and, thus, the phase equilibrium calculation is relatively cheap. In these problems, the flux stencils, the accumulation terms, well tubing flow terms, and well in/out flow terms are

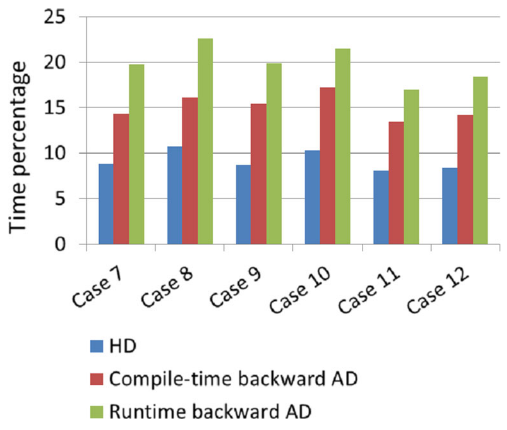

Fig. 6 Time percentages of different differentiation implementations in complete simulations. In these cases, only a part of the lineaization work (usually the difficult part) is done automatically

linearized by the runtime backward AD or the compile-time backward AD, and the variables are substituted automatically. Lower level modules, such as mobilities, potential differences, permeabilities and porosities are programmed with the HD because they are quite simple. So, only a part of the linearization work is done automatically. For each case with each differentiation method, the time spent on this part of the work is recorded during the simulation (with a timer being accurate to microseconds), and its percentage to the total time is calculated, which is plotted in Fig. 6. Percentages of the HD implementation are also included, to serve as references.

We can see that this part of the linearization work only accounts for a small portion of the total time, for most of the time is consumed by the linear solver, table looking and phase equilibrium calculation (compositional model). The performance deterioration mainly comes from the backward AD phase; meanwhile, the variable substitution phase may take a considerable amount of time. As the time ratio between the backward AD and the HD has been improved to a small value (in single digits), the additional time cost incurred by the AD is not obvious for the whole simulation, even with the runtime version.

4 Conclusions

In this work, we present a two-stage framework to differentiate nonlinear equations in reservoir simulations, i.e., the backward mode AD and the automatic variable substitution. According to whether the expression is determined dynamically and whether the output gradient vector is sparse, the initialization and calling of a module are done differently (Fig. 7). It is best to use compile-time backward AD all of the time, but we cannot always avoid dynamically determined expressions. However, it is found that the performance loss is not serious from a global perspective.

Our approach narrows the gap of computational performance between AD and HD, through the following methods: (1) choosing the backward mode to save computational work; (2) shifting to compile-time polymorphism when the expression is statically determined; and (3) accelerating the automatic variable substitution. Moreover, the framework does not break the independence of reservoir simulator modules. When new modules are created with our framework, the legacy code of a simulator is more or less unaffected. The old modules only need to inform the new modules of the sparse patterns of their output gradient vectors.

The performance of the backward mode AD is demonstrated with the illustrative examples. Its FLOPs cost has dropped to an acceptable level, and the compile-time polymorphism reduces the executing time. Although it is still not as fast as the HD, the work of linearization is not the most computationintensive part of reservoir simulations. As demonstrated in Section 3.3, it only occupies a small portion of the overall time. However, introducing AD into simulator design will facilitate great simplification in coding. With the proposed framework, a reservoir simulator can be easily extended and quickly adapted to the ever-changing needs in gas and oil recovery.

Acknowledgments This work is partially funded by the National Science and Technology Major Project of China (Grant Nos. 2011ZX05009-006 and 2011ZX05052), the National Key Technology R&D Program of China (Grant No. 2012BAC24B02), and the National Natural Science Foundation of China (Grant Nos 51204008 and U1262204). We would like to thank the anonymous reviewers and the editors for their insightful, constructive comments, which have led to a much improved version.

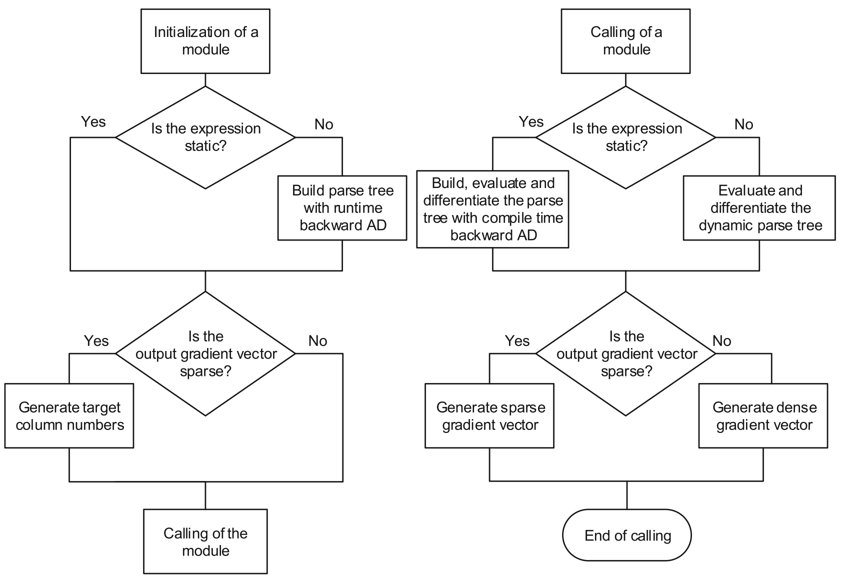
Initialization of a module
Is the expression static?
Yes
No
Build parse tree with runtime backward AD
Is the output gradient vector sparse?
Yes
No
Generate target column numbers
Calling of the module
Calling of a module
Is the expression static?
Yes
No
Build, evaluate and differentiate the parse tree with compile time backward AD
Evaluate and differentiate the dynamic parse tree
Is the output gradient vector sparse?
Yes
No
Generate sparse gradient vector
Generate dense gradient vector
End of calling

Fig. 7 The initialization and calling of a module created by our AD framework

Appendix 1 : A simple sample for the use of compile-time backward AD

```
Leaf a(0), b(1), c(2);    /* Set independent variable indices */
double Varg[3] = {1.0, 2.0, 3.0}, Grad[3] = {0.0, 0.0, 0.0}, Vout;
/* Upon initialization, provide the values of a, b, c and set the gradient vector to be 0 */
#define    f    a*b*c    /* Macro definition for the expression */
Vout = RADEval(f, Varg, Grad);
```

After calling “RADEval”, the value of “Vout” is 6.0, and the values of “Grad” are (6.0, 3.0, 2.0), which are the values of $\left(\frac{\partial f}{\partial a}, \frac{\partial f}{\partial b}, \frac{\partial f}{\partial c}\right)$

As “f” is simple, we can avoid using macro definitions; so, the last two lines of the code can be simplified as:

```
Vout = RADEval(a*b*c, Varg, Grad);
```

Appendix 2: Calculating well index (z-component only) using Peaceman’s formula

```
#include "ADLib.h"
double Ad_WIz(const double rw, const double skin, const double hz,/*Input*/
             const double dx, const double dy,                 /*Input*/
             const double kx, const double ky,                 /*Input*/
             double& dkx, double& dky)  /*Output derivatives*/
{
    using namespace RADTEMP;          /* Namespace of the reverse AD */
    Leaf KX(0), KY(1);               /* Leaf nodes, 0,1 are indices */
    Cnst HZ(hz), DX2(dx*dx), DY2(dy*dy); /* Constant nodes */
    Cnst SKIN(skin), IRW(0.28/rw);     /* Constant nodes */
    double dk[2] = {0.0, 0.0};
    double k[2] = {kx, ky};
    double WI;
                                 /* Define the templated expression */
    #define WI_exp  \
    HZ*ADsqrt(KX*KY)/(ADlog(IRW*ADsqrt(KX*DY2+KY*DX2) \
    /(ADsqrt(KX)+ADsqrt(KY)))+SKIN)
    {
       RADEval(WI_exp, k, WI, dk);   /* Evaluating and differentiating */
    }
    dkx = dk[0];                    /* Export the result */
    dky = dk[1];
    return WI;
};
```

References

1. Alexandrescu, A.: Modern C++ design: generic programming and design patterns applied. Addison-Wesley Professional (2001)
2. Aubert, P., Di Césaré, N., Pironneau, O.: Automatic differentiation in C++ using expression templates and application to a flow control problem. Comput. Vis. Sci. **3**(4), 197–208 (2001)
3. Baur, W., Strassen, V.: The complexity of partial derivatives. Theor. Comput. Sci. **22**(3), 317–330 (1983)
4. Beda, L., Korolev, L., Sukkikh, N., Frolova, T.: Programs for automatic differentiation for the machine BESM. Inst. Precise Mechanics and Computation Techniques. Academy of Science, Moscow (1959)
5. Bendtsen, C., Stauning, O.: FADBAD, a flexible C++ package for automatic differentiation. Department of Mathematical Modelling. Technical University of Denmark (1996)
6. Bischof, C., Khademi, P., Mauer, A., Carle, A.: ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. IEEE Comput. Sci. Eng. **3**(3), 18–32 (1996)
7. Bischof, C.H., Haghighat, M. Hierarchical approaches to automatic differentiation. BBCG96, 83 (1996)
8. Borodin, A., Munro, I.: The computational complexity of algebraic and numeric problems. (1975)

9. Cacuci, D.G.: Sensitivity theory for nonlinear systems. I. Nonlinear functional analysis approach. J. Math. Phys. **22**(12), 2794–2802 (1981)
10. Cacuci, D.G.: Sensitivity theory for nonlinear systems. II. Extensions to additional classes of responses. J. Math. Phys. **22**(12), 2803–2812 (1981)
11. Cao, H.: Development of techniques for general purpose simulators. Stanford University (2002)
12. DeBaun, D., Byer, T., Childs, P., Chen, J., Saaf, F., Wells, M., Liu, J., Cao, H., Pianelo, L., Tilakraj, V.: An extensible architecture for next generation scalable parallel reservoir simulation. In: SPE Reservoir Simulation Symposium (2005)
13. Griewank, A.: On automatic differentiation. Mathematical Programming: recent developments and applications **6**, 83–107 (1989)
14. Haaland, S.E.: Simple and explicit formulas for the friction factor in turbulent pipe flow. J. Fluids Eng.; (United States) **105**(1) (1983)
15. Hogan, R.J.: Fast reverse-mode automatic differentiation using expression templates in C+. Submitted to ACM Trans. Math. Softw. (2014)
16. Huang, H., Ayoub, J.: Applicability of the Forchheimer equation for non-Darcy flow in porous media. SPE J. **13**(1), 112–122 (2008)
17. Jiang, Y.: Techniques for modeling complex reservoirs and advanced wells. Stanford University (2007)
18. Lie, K.A., Krogstad, S., Ligaarden, I.S., Natvig, J.R., Nilsen, H.M., Skaflestad, B.: Open-source MATLAB implementation of consistent discretisations on complex grids. Comput. Geosci. **16**(2), 297–322 (2012)
19. Lieberherr, K.J., Holland, I.M.: Assuring good style for object-oriented programs. IEEE Softw. **6**(5), 38–48 (1989)
20. Peaceman, D.W.: Interpretation of well-block pressures in numerical reservoir simulation. Soc. Pet. Eng. J. **18**(03), 183–194 (1978)
21. Phipps, E., Pawlowski, R.: Efficient expression templates for operator overloading-based automatic differentiation. In: Recent Advances in Algorithmic Differentiation, pp. 309-319. Springer (2012)
22. Phipps, E.T., Bartlett, R.A., Gay, D.M., Hoekstra, R.J.: Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In: Advances in automatic differentiation, pp. 351-362. Springer (2008)
23. Rall, L.B.: Automatic differentiation: techniques and applications (1981)
24. Shi, H., Holmes, J., Durlofsky, L., Aziz, K., Diaz, L., Alkaya, B., Oddie, G.: Drift-flux modeling of two-phase flow in wellbores. SPE J. **10**(1), 24–33 (2005)
25. Speelpenning, B.: Compiling fast partial derivatives of functions given by algorithms. In. Illinois Univ., Urbana (USA). Dept. of Computer Science (1980)
26. Utke, J.: OpenAD: Algorithm implementation user guide. Technical Mem (2004)
27. 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 Trans. Math. Softw. (TOMS) **34**(4), 18 (2008)
28. Veldhuizen, T.: Expression templates. C++ Report **7**(5), 26–31 (1995)
29. Veldhuizen, T.: Arrays in blitz++. In: Computing in object-oriented parallel environments, pp. 223-230. Springer (1998)
30. Wei, Z., Zhang, D.: Coupled fluid-flow and geomechanics for triple-porosity/dual-permeability modeling of coalbed methane recovery . International Journal of Rock Mechanics and Mining Sciences **47**(8), 1242–1253 (2010)
31. Wengert, R.: A simple automatic derivative evaluation program. Commun. ACM **7**(8), 463–464 (1964)
32. Wolfe, P.: Checking the calculation of gradients. ACM Trans. Math. Softw. (TOMS) **8**(4), 337–343 (1982)
33. Younis, R., Aziz, K.: Parallel automatically differentiable data-types for next-generation simulator development. In: SPE Reservoir Simulation Symposium (2007)
34. Younis, R.M.: Modern advances in software and solution algorithms for reservoir simulation. Stanford University (2011)
35. Zhou, Y., Tchelepi, H., Mallison, B.: Automatic differentiation framework for compositional simulation on unstructured grids with multi-point discretization schemes. In: SPE Reservoir Simulation Symposium (2011)