Noname manuscript No.
(will be inserted by the editor)

Capitalizing on *Live* Variables: New Algorithms for Efficient Hessian Computation via Automatic Differentiation

Mu Wang · Assefaw Gebremedhin · Alex Pothen

Received: date / Accepted: date

Abstract We revisit an algorithm (called Edge Pushing (EP)) for computing Hessians using Automatic Differentiation (AD) recently proposed by Gower and Mello (2012). Here we give a new, simpler derivation for the EP algorithm based on the notion of live variables from data-flow analysis in compiler theory and redesign the algorithm with close attention to general applicability and performance. We call this algorithm LIVARH and develop an extension of LIVARH that incorporates preaccumulation to further reduce execution time—the resulting algorithm is called LIVARHACC. We engineer robust implementations for both algorithms LIVARH and LIVARHACC within ADOL-C, a widely-used operator overloading based AD software tool. Rigorous complexity analyses for the algorithms are provided, and the performance of the algorithms is evaluated using a mesh optimization application and several kinds of synthetic functions as testbeds. The results show that the new algorithms outperform state-of-the-art sparse methods (based on sparsity pattern detection, coloring, compressed matrix evaluation, and recovery) in some cases by orders of magnitude. We have made our implementation available online as open-source software and it will be included in a future release of ADOL-C.

Keywords Algorithmic Differentiation · Hessian Computation · Reverse Mode AD · Edge Pushing · ADOL-C · Data-flow Analysis.

Mu Wang
Department of Computer Science, Purdue University, West Lafayette, IN 47907
E-mail: wang970@purdue.edu

Assefaw Gebremedhin
School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA 99164
E-mail: assefaw@eecs.wsu.edu

Alex Pothen
Department of Computer Science, Purdue University, West Lafayette, IN 47907
E-mail: apothen@purdue.edu

Mathematics Subject Classification (2000) 90C30 (Nonlinear programming) · 49M37 (Methods of nonlinear programming type) · 65K05 (Mathematical programming methods)

Contents

1 Introduction

We deal with the design, analysis, implementation and performance evaluation of algorithms for the efficient computation of Hessians using Automatic, or Algorithmic, Differentiation (AD).

AD Background. AD is a technology for analytically (thus accurately within machine precision) computing the derivatives of a function encoded as a computer program. AD relies on the premise that the execution of any computer-coded function, regardless of how complex it is, can be decomposed into a finite sequence of elementary functions—operators or intrinsic functions. Once decomposed, the derivatives of each elementary function can be directly computed, and then the *chain rule of calculus* is applied on the sequence to yield the derivatives of the objective function.

An objective, or input, function could in general be either *scalar* or *vector*. Since our focus here is on Hessians, we will be concerned with the former. Let $\mathbf{y} = \mathbf{F}(\mathbf{x})$, $\mathbf{x} \in \mathcal{R}^{\mathbf{n}\times\mathbf{1}}$ and $\mathbf{y} \in \mathcal{R}$ denote a scalar objective function of real values. Following the notations of Griewank and Walther [1], both here and elsewhere in the paper, let $v_{1-n}, \cdots, v_0$ denote the *independent variables* ($\mathbf{x}$). The execution of the objective function can then be decomposed as:

$$\begin{aligned} &\textbf{for } i = 1, 2, \cdots, l \\ &\qquad v_i = \varphi_i(v_j)_{v_j \prec v_i, 1-n \leq j < i}, \end{aligned}$$

where each $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ represents the evaluation of an elementary function, and $v_j \prec v_i$ denotes that variable v_i directly *depends* on variable v_j. That is, in each step, v_i is the result of the elementary function φ_i, which takes v_j where $v_j \prec v_i$ as operands or parameters (for intrinsics). We call each $v_i = \varphi_i(v_j)$ a *Single Assignment Code* (SAC)[1]. The last variable in the

[1] The term Single Assignment Code is used in Griewank-Walther [1] to refer to a block of evaluation procedure rather than an elementary function; here we use it to mean the latter, since, for simplicity, we identify mathematical variables with program variables.

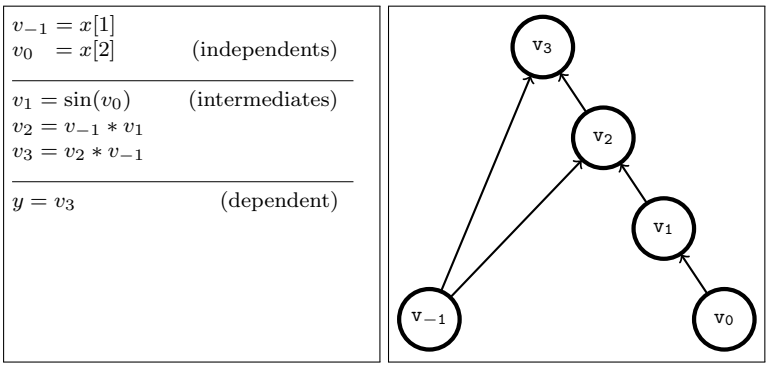


Fig. 1 The relationship between a function, its SAC and the corresponding computational graph. The illustration is for the function given in Example 1. The SAC sequence is shown in the left box, and the computational graph is depicted in the right box.

sequence of elementary functions, v_l, holds the return value of the objective function, the only *dependent variable*. Evaluating the objective function **F** is, thus, equivalent to evaluating the SAC sequence.

The evaluation of an objective function in the above fashion can be modeled using a *directed acyclic graph*, where vertices represent the variables and edges represent the precedence relations. Such a graph is known in the AD literature as the *computational graph* of the function.

Example 1 Consider a function defined by the line of code $y = \big(x[1] * \sin(x[2])\big) * x[1]$. We will use this function as a running example to illustrate various points throughout the paper.

Figure 1 shows how the function given in Example 1 is represented in terms of SACs and an associated computational graph.

Traditional Hessian Algorithms. For a given scalar objective function, the corresponding computational graph contains sufficient structural information for the purpose of evaluating the function's first order derivative, the gradient. This statement is true whether the computation is done via the *forward mode* or the *reverse mode* of AD. In particular, what needs to be done is to associate appropriate derivative quantities (tangents in the case of forward mode and adjoints in the case of the reverse mode) with the vertices of the graph and associate weights with the edges of the graph to represent partial derivatives.

For Hessian computation, on the other hand, the computational graph of the function by itself is insufficient, since it does not capture nonlinear interactions between variables.

Traditional AD algorithms for Hessian, either implicitly or explicitly, work with some "extended" version of the computational graph to deal with this deficiency. Depending on the way in which the forward and the reverse mode

are combined to obtain the second order derivatives, these traditional approaches can potentially give rise to four different methods: forward-over-forward, forward-over-reverse, reverse-over-forward, and reverse-over-reverse. Of these four, forward-over-forward and reverse-over-reverse can be easily ruled out since they are obviously impractical or highly inefficient. Of the remaining two, forward-over-reverse, which is also called second-order-adjoint mode (SOAM), is known to be significantly more efficient than reverse-over-forward [1,2].

A SOAM-based approach works with a *Hessian computational graph* that consists of two distinct parts: the computational graph of the objective function and the computational graph of the adjoints. The graph of the adjoints is a mirror copy of the graph of the objective function, with the direction between vertices reversed to reflect that the dependency between adjoints is in the opposite direction relative to the dependency between intermediate variables. Further, nonlinear interactions are captured by edges joining the two parts.

The Hessian computational graph that results from this abstraction is symmetric. Several previous studies (within the basic framework of SOAM) have focused on detecting and exploiting this symmetry, but most of them fall short in their ability to exploit the symmetry to its fullest performance benefit [3–6]. To be precise, a SOAM method actually computes a Hessian-vector product as a basic unit, and the full Hessian is then obtained via multiple Hessian-vector products. Therefore, the approach is inherently limited in its capacity to dynamically and fully take advantage of the symmetry in the Hessian computational graph. Consequently, a SOAM-based method typically relies on a rather static exploitation of sparsity, the sparsity that is available in the ultimate Hessian matrix rather than the computational graph of the Hessian. This is done via the *compression-and-recovery* paradigm, a paradigm equally applicable to automatic differentiation as it is to finite differencing [7–10].

The Edge Pushing Algorithm. Gower and Mello [11], recently introduced a fundamentally different graph model for Hessian computation using the reverse mode of AD. In their model, the computational graph of the function is augmented with minimal information, exploiting symmetry. The minimal augmenting information (to the computational graph of the function) is additional edges corresponding precisely with the nonlinear interactions. Using the graph model, Gower and Mello developed an algorithm for Hessian computation they named *Edge Pushing*, where edges corresponding to nonlinear interactions are "created" and their contributions to descendants are "pushed" as the algorithm proceeds. In a follow-up work, they develop a variant of the algorithm that computes only the sparsity pattern of the Hessian and show that the method results in better performance than previously known Hessian sparsity pattern detection algorithms [12].

The Edge Pushing algorithm work presented in [11] makes a significant stride towards the construction of efficient, symmetry-exploiting AD algorithms for Hessians. Meanwhile, it also opens up several important avenues

for improvements and corrections. First, the derivation of the algorithm is somewhat complicated. Specifically, the algorithm is derived by viewing the SAC sequences as a composition of multivariable vector functions and then writing down the closed form of the Hessian under that representation. This makes the derivation hard to understand and obscures the concepts central to the algorithm. Second, the implementation of the algorithm based on ADOL-C relies on an indexing assumption that is not necessarily supported by native ADOL-C. As a result, the code gives incorrect results in some cases. Third, the implementation in general places little or no emphasis on performance.

Our Contribution. In this work, we derive the Edge Pushing algorithm from a completely different perspective, and provide a robust implementation that works correctly in all cases. We show that a "better" way to derive the algorithm is to work with just the precedence relations between variables and apply the chain-rule of calculus directly. Nevertheless, we use the computational graph to provide helpful alternative perspectives and to prove auxiliary structural properties. We outline the main contributions of this paper below:

1. **Invariant and Algorithm Design**. As a foundation for our approach, we identify an invariant in the first order *incremental reverse mode* of AD by taking a *data-flow* perspective. We then obtain the Hessian algorithm by extending the invariant to second order. Central to the data-flow perspective is a recognition of the role of *live variables* in reverse mode AD. This perspective not only simplifies the derivation of the Edge Pushing algorithm but it also serves as a useful guide to a correct implementation and as a framework for extension to higher order derivatives. We call the live variable-centered Hessian algorithm LIVARH.
2. **Implementation**. We implement LIVARH in ADOL-C, a widely-used operator overloading based AD software tool [14,15]. In order to take advantage of the symmetry available in Hessian computation, the result variables v_i in the SAC sequence must have monotonic indices. However, the location scheme for variables currently used in native ADOL-C does not satisfy this property, which is one reason why the Gower-Mello implementation of the Edge Pushing algorithm fails. As a part in our implementation, we developed a routine (outside ADOL-C) that translates the indices appropriately. Our entire implementation is made available as open-source at `https://github.com/CSCsw/LivarH`.
3. **Raising Performance via Preaccumulation**. To further improve efficiency, we incorporate statement-level *preaccumulation* in LIVARH and its implementation—we call the resultant algorithm LIVARHACC. Preaccumulation divides the evaluation into local and global components. Its use reduces the number of global operations significantly, and in most cases, it also reduces the total number of operations, resulting in overall performance gain.
4. **Complexity Analysis**. We provide rigorous complexity analysis for both LIVARH and LIVARHACC. We uncover the circumstances and spell out the

conditions under which the gain due to preaccumulation is maximized. One of our complexity results is that the runtime of LIVARH is $O(lq)$, where l is the number of SACs into which the input function is decomposed and q is the maximum size of live variable set attained during the course of the algorithm.

5. **Empirical Performance Evaluation**. Using both synthetic and real-world testcases, we conduct a systematic experimental evaluation of the performance of the new algorithms and compare these against several existing approaches, including sparse methods based on compression-and-recovery. We find that the new algorithms outperform the existing methods in some cases by several orders of magnitude.

The remainder of the paper is structured as follows. We formalize our observation on the invariant in first order derivative computation in Section 2. We extend the observation to the second order in Section 3, and show in the same section how we use that extension to derive the new Hessian algorithms. We discuss preaccumulation and its incorporation in the Hessian algorithm in Section 4. We describe our implementation and associated design decisions in Section 5. We present and discuss performance evaluation results on experiments conducted using mesh optimization and other applications in Section 6. We give concluding remarks and point out avenues for further work in Section 7. In the Appendix, we provide proofs and other details left out from the analyses in Sections 2 thru 6.

2 Incremental Reverse Mode AD

In forward mode AD, derivatives are evaluated in the same order as the evaluation of the intermediate variables of the objective function. To implement this mode, an AD tool evaluates the derivatives simultaneously with the SAC sequence. In contrast, in reverse mode AD, derivatives are evaluated in an order opposite to that of the objective function evaluation. To implement the reverse mode, therefore, pertinent information on the SAC sequence needs to be stored in some internal data structure. This is often called the *trace* of the function. The SAC sequence is then traversed in the reverse order in the trace to evaluate the derivatives.

The reverse mode has an extremely attractive feature: using it, the gradient of a function can be computed at a temporal cost that is a small multiple of the cost of evaluating the function, completely independent of the number of input variables in the objective function [1]. This nice feature comes at the expense of a higher memory cost relative to the requirements in the forward mode. Devising reverse mode-based methods that strike the right balance in memory/run-time trade-off has long been an active research area in AD [1,2]. The Hessian algorithm we describe here is based entirely on the reverse mode without any checkpointing or related memory saving strategies.

Live Variables. In data-flow analysis in compiler theory, a variable is said to be *live* if it holds a value that *might be* read in the future. Clearly, identifying such variables facilitates analysis of the program. We find a similar notion—that takes the advantage even a step further—useful in our context. In particular, since in reverse mode AD all information about the execution sequence of the objective function is recorded on an evaluation trace, we can in fact work with a more restricted definition of a live variable: we say a variable is *live* if it holds a value that *will be* read in the future. We call the set of all live variables at each step of the execution sequence a *live variable set* in that sequence. In the forward mode, we can deduce from the definition that the independent variables are the only live variables initially, and that v_l is the only live variable finally. In the reverse mode, this order of live variable sets would be reversed.

There are two variants of first order reverse mode in AD: incremental and non-incremental [1]. The non-incremental reverse mode is mainly of theoretical interest since it requires global information in processing each SAC. The incremental reverse mode is what is more commonly implemented in practice. Following the notation of Griewank and Walther mentioned earlier in Section 1, the first order incremental reverse mode of AD can be written as follows:

Algorithm: First Order Incremental Reverse Mode (FOIRM)

Input: a function f expressed as a SAC sequence $v_i = \varphi_i(v_j)_{v_j \prec v_i}, 1 \leq i \leq l$

Output: the adjoints $\{\bar{v}_{1-n}, \cdots, \bar{v}_0\} \equiv \{\frac{\partial v_l}{\partial v_{1-n}}, \cdots, \frac{\partial v_l}{\partial v_0}\}$

Initialization:

$\bar{v}_l = 1.0, \quad \bar{v}_{1-n} = \cdots = \bar{v}_0 = \bar{v}_1 = \cdots = \bar{v}_{l-1} = 0$

for $i = l, \cdots, 1$ **do**

  for all $v_j \prec v_i$ **do**

    $\bar{v}_j += \frac{\partial \varphi_i}{\partial v_j} \bar{v}_i$

Note that the loop in the algorithm FOIRM traverses the SAC sequence in the reverse order to that of the function evaluation. Note further that in each step, only one SAC is processed, and information on local partial derivatives is used to compute the adjoints incrementally.

We will now make the definitions of live variables and equivalent functions precise in the context of the FOIRM algorithm. Table 1 provides an example that illustrates the use of live variables in adjoint computation via the FOIRM algorithm. The reader will find it helpful to follow along in this table during the steps of the algorithm. The example uses the same function as that used in Figure 1, but it is written in the equivalent form $v_3 = (v_{-1} * \sin(v_0)) * v_{-1}$. As a result, the SAC sequence in the left column is the same as that in Figure 1. The middle column gives the live variable set and the intermediate results computed in FOIRM. The right column gives the expressions for the functions defined by the processed SACs and their adjoints.

At the beginning of the reverse mode, when no SAC has yet been processed, all we have is the dependent variable v_l. Thus we can define an equivalent function $\mathbf{F}_{l+1}(v_l) = v_l$. In the example, this corresponds to the function $\mathbf{F}_4$, and

Table 1 An example illustrating the use of live variables in adjoint computation using FOIRM (Observation 1). The underlying function is the same function given in Example 1.

Execution sequence in each step (SAC)	**Live variables and adjoints**	**Equivalent obj. function w.r.t live var. and the derivatives**
Initialization:	$S_4 = \{\mathbf{v_3}\}$	$\mathbf{F_4}(S_4) = v_3$
	$\bar{v}_3 = 1$	$\bar{v}_3 = 1$
	$\bar{v}_2 = \bar{v}_1 = \bar{v}_0 = \bar{v}_{-1} = 0$	
$v_3 = v_2 * v_{-1}$	$S_3 = \{\mathbf{v_{-1}}, \mathbf{v_2}\}$	$\mathbf{F_3}(S_3) = v_2 * v_{-1}$
	$\bar{v}_{-1} = 0 + v_2\bar{v}_3 = v_{-1}\sin(v_0)$	$\bar{v}_{-1} = v_2$
	$\bar{v}_2 = 0 + v_{-1}\bar{v}_3 = v_{-1}$	$\bar{v}_2 = v_{-1}$
$v_2 = v_{-1} * v_1$	$S_2 = \{\mathbf{v_{-1}}, \mathbf{v_1}\}$	$\mathbf{F_2}(S_2) = v_{-1}^2 * v_1$
	$\bar{v}_{-1} = v_{-1}\sin(v_0) + v_1\bar{v}_2$	$\bar{v}_{-1} = 2v_{-1}v_1$
	$\quad = 2v_{-1}\sin(v_0)$	
	$\bar{v}_1 = 0 + v_{-1}\bar{v}_2 = v_{-1}^2$	$\bar{v}_1 = v_{-1}^2$
$v_1 = \sin(v_0)$	$S_1 = \{\mathbf{v_{-1}}, \mathbf{v_0}\}$	$\mathbf{F_1}(\mathbf{S_1}) = v_{-1}^2 * \sin(v_0)$
	$\bar{v}_{-1} = 2v_{-1}\sin(v_0)$	$\bar{v}_{-1} = 2v_{-1}\sin(v_0)$
	$\bar{v}_0 = 0 + \cos(v_0)\bar{v}_1$	$\bar{v}_0 = v_{-1}^2\cos(v_0)$
	$\quad = v_{-1}^2\cos(v_0)$	
v_{-1}, v_0 are independents		

live variable set $S_4 = \{v_3\}$. The first SAC to be processed is $v_l = \varphi_l(v_j)_{v_j \prec v_l}$. After this SAC is processed, we have an equivalent function $\mathbf{F}_l$ obtained by replacing v_l in $\mathbf{F}_{l+1}$ by the variables on the right-hand-side in $v_l = \varphi_l(v_j)_{v_j \prec v_l}$. That is, $\mathbf{F}_l(v_j)_{v_j \prec v_l} = \mathbf{F}_{l+1}(v_l = \varphi_l(v_j)_{v_j \prec v_l})$. In the second step in the example, we have replaced v_3 by v_2 and v_{-1}, to obtain the function $\mathbf{F}_3$ and the new set of live variables $S_3 = \{v_2, v_{-1}\}$.

At each step i from l to 1, we inductively define an equivalent function $\mathbf{F}_i$ in this manner. That is, $\mathbf{F}_i$ is defined by treating v_i in $\mathbf{F}_{i+1}$ as a composite function defined by the SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ wherein v_i is considered as an independent variable in $\mathbf{F}_{i+1}$.

The function $\mathbf{F}_1$ is defined on the variables $\{v_{1-n}, \cdots, v_0\}$, since it is the objective function. It is natural to ask what set of variables each $\mathbf{F}_i, 1 < i \leq l+1$ is defined on. At first glance, it might appear that $\mathbf{F}_i$ is defined on $\{v_{1-n}, \cdots, v_{i-1}\}$. On a closer look, however, one can see that not every variable in that set will necessarily be used in later SACs. Specifically, $\mathbf{F}_i$ is defined only on those variables that will indeed be used in later SACs. Those variables constitute precisely the live variable set S_i; hence we can write the incremental formula that defines S_i from S_{i+1}:

$$S_i = \{S_{i+1} \setminus \{v_i\}\} \cup \{v_j | v_j \prec v_i\}. \tag{1}$$

We call the series of functions $\mathbf{F}_i(S_i)$ *equivalent functions*, because they express the objective function v_l in terms of the current live variable set S_i.

We observe that the set of live variables at step i includes not only the variables in the definition of φ_i, but also variables from earlier steps that will be used at some future step. E.g., in the table, S_1 includes v_{-1} although it is not a variable on which v_1 depends. Hence, we can formally define

$$S_i = \{v_k : 1 - n \leq k < i \; s.t. \; \exists \, v_j \text{ with } v_k \prec v_j \text{ and } i \leq j \leq l\}. \tag{2}$$

In the FOIRM algorithm, when we arrive at the point corresponding to $\mathbf{F}_i$ in the SAC sequence, the algorithm has already processed the SACs $\varphi_l, \ldots, \varphi_i$. The current state can be viewed as an intermediate result if we want to compute the adjoints for $\mathbf{F}_1$, which is defined by the whole original SAC sequence. *However*, the current state can also be viewed as exactly the adjoints of $\mathbf{F}_i$, which is defined by the SACs that have already been processed. The set of independent variables of $\mathbf{F}_i$ is the live variables set S_i. Thus we can state an invariant of the FOIRM algorithm as follows.

Observation 1 *The set of adjoints computed at the end of each step i of FOIRM constitute the partial derivatives with respect to only the current live variable set S_i, not the entire set of variables.*

The notion of live variables and its usage here bears some similarity to the To-Be-Recorded (TBR) analysis studied, for example, in [16,17]. In both TBR analysis and the live variables analysis introduced here, the ideas originate from data-flow analysis in compilers. In that they are similar, but the two differ in how they are applied. TBR analysis is done by analyzing the call graph of a code, so it is necessarily static and conservative. In contrast, live variables analysis is based on a specific execution path of a code. For example, given a conditional statement, TBR analysis would need to consider both branches of the statement. In contrast, in the live variables analysis used here, only the information of the executed branch is recorded (on the trace) and judiciously retrieved for effective use. Furthermore, TBR analysis aims at a coarser granularity, targeting constructs such as basic blocks or even functions, whereas live variable analysis targets elementary functions. Therefore, TBR analysis is better suited for AD tools based on source code transformation, whereas live variable analysis is well-suited for operator overloading based AD, since it works at the level of elementary functions.

3 The Hessian Algorithm

We now extend the invariant in first order incremental reverse mode expressed in Observation 1 to the second order. This extension is an initial step we need to derive our target Hessian algorithm.

3.1 The Second Order Invariant

Stated formally the invariant reads:

Observation 2 *At the completion of the processing of the i-th SAC, the proposed Hessian algorithm maintains the Hessian of the equivalent function* $\mathbf{F}_i$ *with respect to the current live variable set* S_i*.*

For a given equivalent function $\mathbf{F}_i(S_i)$, the first order derivatives (adjoints), which we previously denoted by $\bar{v}$, for each $v \in S_i$, can be viewed as a mapping

$$a_i : S_i \to \mathbb{R},\ a_i(v) = \bar{v} = \frac{\partial \mathbf{F}_i}{\partial v}. \tag{3}$$

(We use the notation $a_i(v)$ instead of $\bar{v}$ to make it easier to see the connections.)

Extending the mapping, the second order derivatives (Hessian) for the equivalent function $F_i(S_i)$ can be viewed as a symmetric mapping

$$h_i : S_i \times S_i \to \mathbb{R},\ h_i(v,u) = h_i(u,v) = \frac{\partial a_i(v)}{\partial u}. \tag{4}$$

The target algorithm is then precisely a prescription of how to compute S_i, $a_i(S_i)$ and $h_i(S_i, S_i)$ to maintain the second order invariant.

Suppose the next SAC to be processed is $v_i = \varphi_i(v_j)_{v_j \prec v_i}$. Before processing this SAC, according to the invariant, we have a current live variable set S_{i+1}, an equivalent function $\mathbf{F}_{i+1}(S_{i+1})$ and a current adjoint mapping $a_{i+1}(S_{i+1})$ as well as a Hessian mapping $h_{i+1}(S_{i+1}, S_{i+1})$ for the equivalent function $\mathbf{F}_{i+1}(S_{i+1})$. After this SAC is processed, we will have a new current live variable set S_i, a new equivalent function $\mathbf{F}_i(S_i)$ and a new adjoint mapping $a_i(S_i)$ as well as a new Hessian mapping $h_i(S_i, S_i)$ for the equivalent function $\mathbf{F}_i(S_i)$. So what we need to establish is the relationship between $\{S_i, a_i(S_i), h_i(S_i, S_i)\}$ and $\{S_{i+1}, a_{i+1}(S_{i+1}), h_{i+1}(S_{i+1}, S_{i+1})\}$.

Recall from the last section that the live variable set is updated as

$$S_i = \{S_{i+1} \setminus \{v_i\}\} \cup \{v_j | v_j \prec v_i\}.$$

For adjoints $a_i(S_i)$, from the first order chain rule, we know that

$$\forall v_j \in S_i :\ a_i(v_j) = \begin{cases} \frac{\partial \varphi_i}{\partial v_j} a_{i+1}(v_i), & v_j \prec v_i \text{ and } v_j \notin S_{i+1}, \\ a_{i+1}(v_j) + \frac{\partial \varphi_i}{\partial v_j} a_{i+1}(v_i), & v_j \prec v_i \text{ and } v_j \in S_{i+1}, \\ a_{i+1}(v_j), & v_j \not\prec v_i. \end{cases} \tag{5}$$

Noting that $\frac{\partial \varphi_i}{\partial v_j} = 0$ when $v_j \not\prec v_i$, and $a_{i+1}(v_j) = 0$ when $v_j \notin S_{i+1}$, Expression (5) can be simplified as

$$\forall v_j \in S_i :\ a_i(v_j) = a_{i+1}(v_j) + \frac{\partial \varphi_i}{\partial v_j} a_{i+1}(v_i). \tag{6}$$

This is the same as the reverse mode computation of adjoints given earlier (FOIRM).

In the second order case, assuming that $h_{i+1}(v_j, v_k) = 0$ when $v_j \notin S_{i+1}$ or $v_k \notin S_{i+1}$, and applying the second order chain rule of calculus, we have $\forall v_j, v_k \in S_i$:

$$h_i(v_j, v_k) = \begin{cases} h_{i+1}(v_j, v_k), & v_j \not\prec v_i, v_k \not\prec v_i, \\ h_{i+1}(v_j, v_k) + \frac{\partial \varphi_i}{\partial v_j} h_{i+1}(v_i, v_k), & v_j \prec v_i, v_k \not\prec v_i, \\ h_{i+1}(v_j, v_k) + \frac{\partial \varphi_i}{\partial v_j} h_{i+1}(v_i, v_k) + \frac{\partial \varphi_i}{\partial v_k} h_{i+1}(v_j, v_i) & \\ \quad + \frac{\partial \varphi_i}{\partial v_j} \frac{\partial \varphi_i}{\partial v_k} h_{i+1}(v_i, v_i) + \frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} a_{i+1}(v_i), & v_j \prec v_i, v_k \prec v_i. \end{cases} \quad (7)$$

Analogous to the adjoint case, we can simplify Expression (7) as

$$\begin{aligned} &\forall v_j, v_k \in S_i : \\ &\quad h_i(v_j, v_k) = h_{i+1}(v_j, v_k) \\ &\qquad + \frac{\partial \varphi_i}{\partial v_k} h_{i+1}(v_i, v_j) + \frac{\partial \varphi_i}{\partial v_j} h_{i+1}(v_i, v_k) + \frac{\partial \varphi_i}{\partial v_j} \frac{\partial \varphi_i}{\partial v_k} h_{i+1}(v_i, v_i) \\ &\qquad + \frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} a_{i+1}(v_i). \end{aligned} \quad (8)$$

3.2 A Basic Version of The Hessian Algorithm

Based on what we have developed so far, we can write down an algorithm which directly "implements" Equations (6) and (8). This straightforward Hessian algorithm is given in Figure 2. The algorithm processes the SAC sequence in the reverse order. In the algorithm, instead of maintaining all the sets $\{S_i, a_i(S_i), h_i(S_i, S_i)\}$, we maintain only one set $\{S, a(S), h(S, S)\}$ to keep the current result in such a way that the invariant in Observation 2 is maintained after the SAC is processed. Therefore in subsequent discussions, we exclude the subscripts and simply use $\{S, a(S), h(S, S)\}$. From the invariant, the mapping $h(S, S)$ represents a symmetric matrix, which is the current Hessian matrix with respect to the current live variable set S. Each row/column of the matrix is associated with a current live variable $v \in S$. We will use these two views—the mapping and the matrix—interchangeably in our discussions in this section and elsewhere in future sections.

The processing of each SAC φ_i consists of three substeps. The first substep is to update the live variable set S. From the definition of live variables, we can infer that v_i is *not* a live variable after the processing of the SAC φ_i. Therefore all values in a and h related to v_i need to be removed. But those values are needed in calculating later updates. Thus we use a temporary variable w to store the adjoint $a(v_i)$ and a temporary mapping $r(S)$ to store the row $h(v_i, S)$ corresponding to v_i. The latter is a retrieval followed by a deletion of a row (and a column) in a symmetric matrix. Furthermore, after the processing of the SAC, the new live variable set S will contain all v_j such that $v_j \prec v_i$. Therefore, the values of $a(v_j)$ and $h(v_j, S)$ for those newly added live variables

Algorithm: Basic Reverse Mode Hessian
Input: a function f expressed as a SAC sequence $v_i = \varphi_i(v_j)_{v_j \prec v_i}, 1 \leq i \leq l$
Output: the gradient ∇f as $a(S)$, the Hessian $\nabla^2 f$ as $h(S, S)$
Initialization:
$S = \{v_l\}, \quad a(v_l) = 1.0, \quad h(v_l, v_l) = 0.0$
for $i = l, \cdots, 1$ **do** % Process $v_i = \varphi_i(v_j)_{v_j \prec v_i}$
% Update the live variables S
Set $w = a(v_i)$
Retrieve mapping $r : S \to \mathbb{R}$ as $r(v) = h(v_i, v)$
Remove v_i from S: $a(v_i) = 0, \quad h(v_i, S) = 0, \quad h(S, v_i) = 0$
for all $v_j \prec v_i$ **do**
if $v_j \notin S$
Set $S = S \cup \{v_j\}, \quad a(v_j) = 0, \quad h(v_j, S) = 0$
% Update the adjoints $a(S)$
for all $v_j \prec v_i$ **do**
$a(v_j) = a(v_j) + \frac{\partial \varphi_i}{\partial v_j} w$
% Update the Hessian $h(S, S)$
for all unordered pairs (v_j, v_k) in S such that $v_j \prec v_i$ or $v_k \prec v_i$ **do**
$h(v_j, v_k) = h(v_j, v_k) + \frac{\partial \varphi_i}{\partial v_k} r(v_j) + \frac{\partial \varphi_i}{\partial v_j} r(v_k) + \frac{\partial \varphi_i}{\partial v_j} \frac{\partial \varphi_i}{\partial v_k} r(v_i) + \frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} w$

Fig. 2 A Basic Version of The Hessian Algorithm.

v_j need to be initialized to zero. The first substep thus consists in all of these tasks, and is reflected in the code fragment grouped under the "% Update the live variables S" comment in Figure 2.

The second and third substeps correspond to updating the adjoint mapping $a(S)$ and the Hessian mapping $h(S, S)$ (recall Equations (6) and (8)), respectively. The values w and $r(S)$ stored in the first substep are used here. The derivatives for φ_i are computed directly because φ_i is an elementary function. In the outline given in Figure 2, the second substep (adjoint update) is performed before the third substep (Hessian update). But these substeps could also be swapped; that is, we could update the Hessian first and then update the adjoints. The order will not affect the result.

Finally, note that only the necessary entries are updated as each SAC is processed in the algorithm. For example, in updating the adjoints mapping, we compute the value of $a(v_j)$ only for the variables $v_j \prec v_i$, and in updating the Hessian mapping, we only compute the values of the elements $h(v_j, v_k)$ that arise from the variables $v_j \prec v_i$ or $v_k \prec v_i$. These elements correspond to the elements in the row $r(S)$ and w. That way, the sparsity of the Hessian is automatically exploited.

Figure 3 illustrates the workings of the algorithm using the example objective function we saw in Example 1. The figure gives step-by-step results on the values of the adjoints and Hessian mapping.

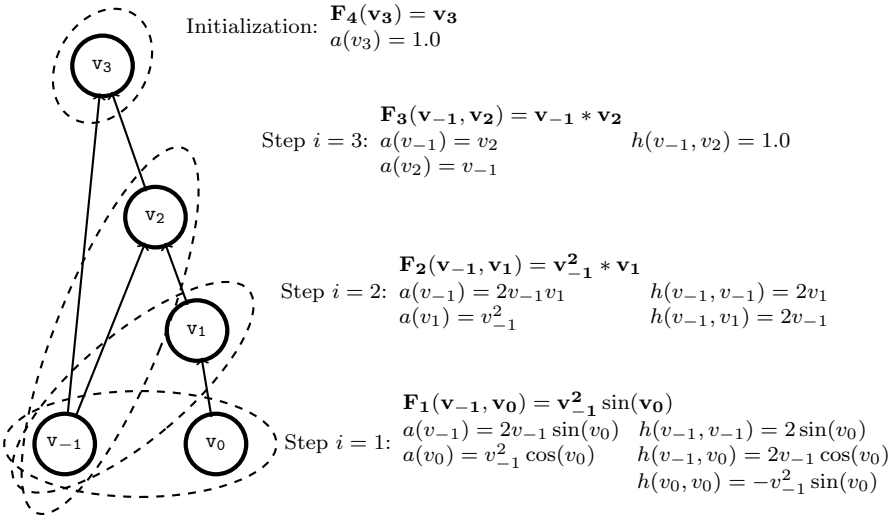


Fig. 3 A step-by-step illustration of how the basic reverse Hessian algorithm outlined in Figure 2 works. The objective function is the same as that in Example 1. In each step, the live variable set is shown enclosed in an eclipse. Notice that in each step, we have the Hessian of the equivalent objective function with respect to the current live variable set.

3.3 An Enhanced Version of The Hessian Algorithm (LIVARH)

The algorithm in Figure 2 does the job, but it is not efficient enough in updating the Hessian mapping. For example, given an unordered pair (v_j, v_k) where $v_j \prec v_i$ and $v_k \in S$, we need to update the entry $h(v_j, v_k)$ by adding to it the four terms:

$$\frac{\partial \varphi_i}{\partial v_k} r(v_j) \ , \ \frac{\partial \varphi_i}{\partial v_j} r(v_k) \ , \ \frac{\partial \varphi_i}{\partial v_j}\frac{\partial \varphi_i}{\partial v_k} r(v_i) \text{ and } \frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} w. \tag{9}$$

Notice, however, that if $r(v_j) = 0$, computing and then adding the value of the term $\frac{\partial \varphi_i}{\partial v_k} r(v_j)$ is unnecessary. Furthermore, computing the values of the terms $\frac{\partial \varphi_i}{\partial v'_k} r(v_j)$ are also unnecessary when updating for other pairs $(v_j, v'_k), v'_k \in S$. Similarly, if $r(v_i) = 0$, we don't need to consider the term $\frac{\partial \varphi_i}{\partial v_j}\frac{\partial \varphi_i}{\partial v_k} r(v_i)$ for all (v_j, v_k).

Therefore, the algorithm can be improved by performing the updates *incrementally*. That is, instead of focusing on each $h(v_j, v_k)$, we focus on the row $r(S)$ and on w. In other words, we carry out only necessary updates and do so at a *finer granularity*. In particular,

- If $r(v_j) \neq 0$, we incrementally update $h(v_j, v_k)$ for variables $v_k \in S, v_k \neq v_j$ by adding $\frac{\partial \varphi_i}{\partial v_k} r(v_j)$ to it, and update $h(v_j, v_j)$ by adding $2\frac{\partial \varphi_i}{\partial v_j} r(v_j)$ to it.
- If $r(v_i) \neq 0$, we add $\frac{\partial \varphi_i}{\partial v_j}\frac{\partial \varphi_i}{\partial v_k} r(v_i)$ to $h(v_j, v_k)$ for $v_j \prec v_i$ and $v_k \prec v_i$.

– If $w \neq 0$, we add $\frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} w$ to $h(v_j, v_k)$ for $v_j \prec v_i$ and $v_k \prec v_i$.

Putting these together, we get the enhanced version of the reverse Hessian algorithm outlined in Figure 4. We refer to this algorithm as LIVARH (short for live variable based Hessian algorithm). LIVARH is equivalent to the Edge Pushing algorithm of Gower and Mello [11], where the updates in the first two bullet items above correspond to the *pushing* part and the last item corresponds to the *creating* part in the component-wise version of their algorithm. Figure 5 shows how these various updates can be interpreted as insertion and manipulation of weighted edges in the computational graph, which is where the notions of pushing and creating originate from.

Algorithm: Enhanced Reverse Mode Hessian (LIVARH)
Input: a function f expressed as a SAC sequence $v_i = \varphi_i(v_j)_{v_j \prec v_i}, 1 \leq i \leq l$
Output: the gradient ∇f as $a(S)$, the Hessian $\nabla^2 f$ as $h(S, S)$

Initialization:
$S = \{v_l\}, \quad a(v_l) = 1.0, \quad h(v_l, v_l) = 0.0$

for $i = l, \cdots, 1$ **do** % Process $v_i = \varphi_i(v_j)_{v_j \prec v_i}$
% Update the live variables S
Set $w = a(v_i)$
Retrieve mapping $r : S \to \mathbb{R}$ as $r(v) = h(v_i, v)$
Remove v_i from S: $a(v_i) = 0, \quad h(v_i, S) = 0, \quad h(S, v_i) = 0$
for all $v_j \prec v_i$ **do**
if $v_j \notin S$
Set $S = S \cup \{v_j\}, \quad a(v_j) = 0, \quad h(v_j, S) = 0$
% Update the adjoints $a(S)$
if $w \neq 0$
for all $v_j \prec v_i$ **do**
$a(v_j) = a(v_j) + \frac{\partial \varphi_i}{\partial v_j} w$
% Update the Hessian $h(S, S)$
% Pushing
for all $v_j \neq v_i$ such that $r(v_j) \neq 0$ **do** % Fig. 5 (a)
for all $v_k \neq v_j$ such that $\frac{\partial \varphi_i}{\partial v_k} \neq 0$ **do**
$h(v_j, v_k) += \frac{\partial \varphi_i}{\partial v_k} r(v_j)$
if $\frac{\partial \varphi_i}{\partial v_j} \neq 0$ % $v_j = v_k$
$h(v_j, v_j) += 2 \frac{\partial \varphi_i}{\partial v_j} r(v_j)$
if $r(v_i) \neq 0$ % Fig. 5 (b)
for all unordered pairs (v_j, v_k) such that $\frac{\partial \varphi_i}{\partial v_j} \frac{\partial \varphi_i}{\partial v_k} \neq 0$ **do**
$h(v_j, v_k) += \frac{\partial v_i}{\partial v_j} \frac{\partial v_i}{\partial v_k} r(v_i)$
% Creating
if $w \neq 0$ % Fig. 5 (c)
for all unordered pairs (v_j, v_k) where $\frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} \neq 0$ **do**
$h(v_j, v_k) += \frac{\partial^2 v_i}{\partial v_j \partial v_k} w$

Fig. 4 Enhanced Version of The Hessian Algorithm (LIVARH).

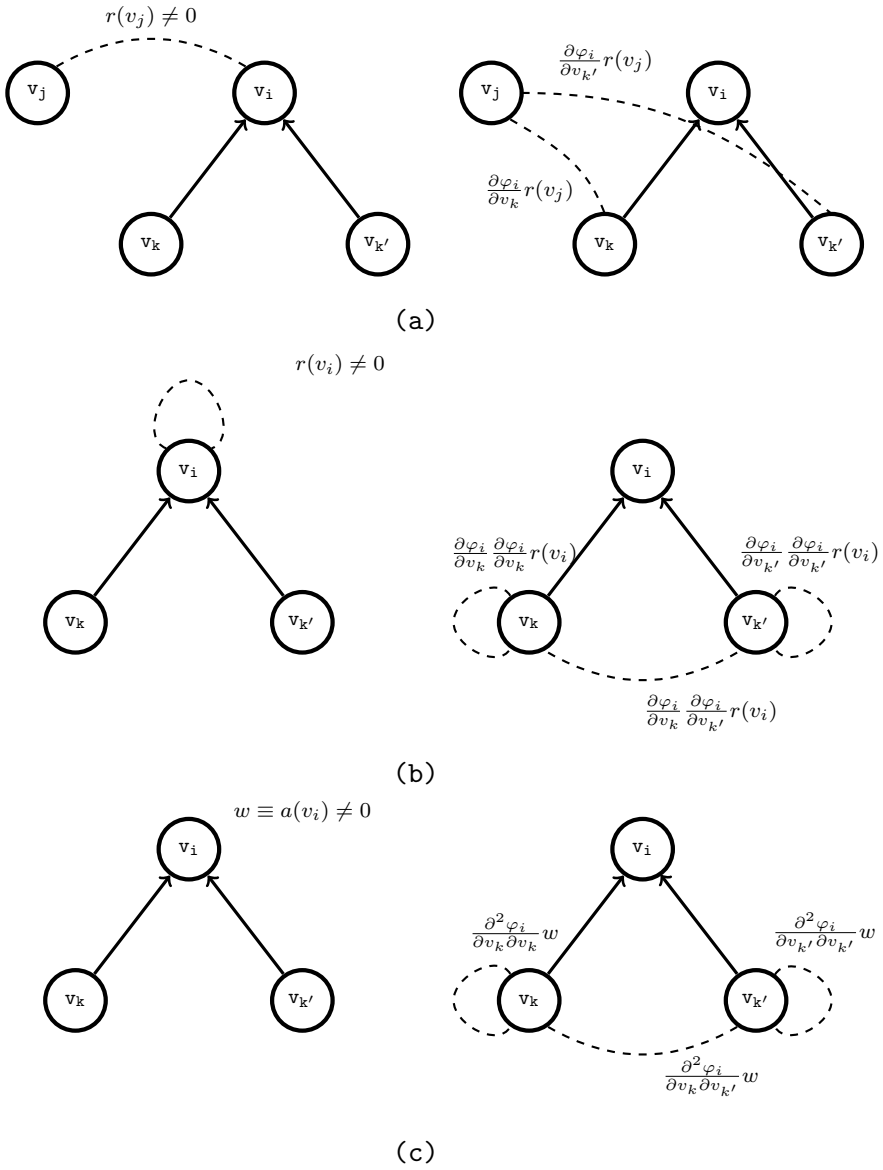


Fig. 5 An example illustrating how the Hessian updates can be represented on the computational graph. The left subfigures show the status before the processing of the SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ and the right subfigures show the status after. A weighted dashed arc between v_i and v_j represents the entry $h(v_i, v_j)$. Subfigure `(a)` represents the updates caused by a nonzero $r(v_j)$. Subfigure `(b)` represents the updates caused by a nonzero $r(v_i)$. Subfigure `(c)` represents the updates caused by a nonzero $w \equiv a(v_i)$.

3.4 Complexity Analysis

We use the number of updates needed of the Hessian mapping as a measure of the time complexity of LIVARH. This number is a good measure for the complexity of the algorithm because the cost of updating the live variable set and the adjoints mapping constitutes a much smaller portion that we can afford to ignore in the analysis.

We make several assumptions to simplify the analysis. In LIVARH, the number of necessary updates on the Hessian mapping depends not only on the structure of the computational graph, but also on the type of each node. For example, an addition operation will not trigger the "edge creating" part in updating the Hessian mapping, because $\frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} = 0$, but a multiplication operation will do so. In our analysis, we compute an upper bound on such numbers. That is, we make the conservative assumption that all local derivatives are nonzero so that the dependency in the computational graph always leads to nonzero derivative values.

First let's consider the number of operations needed to process one SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$. Let the live variable set at this time be S. Let d_i denote the number of nonzero elements in the mapping $r(S)$; that is, $d_i = |\{v_j | r(v_j) \equiv h(v_i, v_j) \neq 0,\ v_j \in S\}|$. Further, let P_i be the set of predecessors of v_i, that is, $P_i = \{v_j | v_j \prec v_i\}$, and let $p_i = |P_i|$. Then we can make the following statements:

(1) Recall that computing the mapping $r : S \to \mathbb{R}$ involves retrieval and deletion of the row/column corresponding to v_i from a symmetric matrix. So, d_i operations are needed for removing the nonzero elements.
(2) In the "pushing" parts, for each $v_j \in S$ and $v_j \neq v_i$, whenever $r(v_j) \neq 0$, we need to enumerate $v_k \in P_i$ and update $h(v_j, v_k)$ (see Figure 5 a). Moreover, whenever $r(v_i) \neq 0$, we need to enumerate all unordered pairs (v_j, v_k) over P_i (see Figure 5 b). Thus, in total, we can see at most $d_i p_i + \frac{p_i(p_i+1)}{2}$ updates are needed.
(3) In the "creating" part, whenever $w \neq 0$, we need to generate all unordered pairs over P_i (see Figure 5 c). So, at most $\frac{p_i(p_i+1)}{2}$ updates are needed.

Thus, overall, we will need at most $(d_i + p_i)(p_i + 1)$ updates to process the SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$. Note that the cost depends on two factors: how complex the SAC is, which is gauged by p_i; and how many non-linear interactions the result variable involves, which is gauged by d_i.

Then by simply taking a summation over all SACs, we get the following upper bound on the complexity of LIVARH:

$$\sum_{i=1}^{i=l} (d_i + p_i)(p_i + 1). \tag{10}$$

In Eq. (10), p_i is the number of operands that the SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ has. Since most elementary functions are unary or binary, we have $p_i = O(1)$. Moreover, d_i is always bounded by the size of the live variable set S existing

right before the processing of $v_i = \varphi_i(v_j)_{v_j \prec v_i}$. Let the maximum size of a live variable set in the course of LIVARH be q. Then we can bound the quantity in Expression (10) as:

$$\sum_{i=1}^{i=l}(d_i + p_i)(p_i + 1) = O(\sum_{i=1}^{i=l}(d_i + p_i)) = O(lq). \tag{11}$$

Expression (11) tells us that the complexity of LIVARH is proportional to the product the number of SACs times the maximum size of live variable set over the course of the algorithm. Note that each of these factors is a property solely of the computational graph of the function.

If we take $d = \max_{i=1}^{i=l}\{d_i\}$, then we can get a better bound:

$$\sum_{i=1}^{i=l}(d_i + p_i)(p_i + 1) = O(\sum_{i=1}^{i=l}(d_i + p_i)) = O(ld). \tag{12}$$

Expression (12) in turn tells us that the complexity of LIVARH is proportional to the product the number of SACs times the maximum number of nonzeros per row in all of the Hessian matrices (mappings) occurring in the course of the algorithm. This shows that the algorithm exploits the sparsity available in the Hessian matrix.

We note that Gower and Mello have given an analysis of the complexity for their Edge Pushing algorithm [11,12]. The complexity expression they obtained using a similar measure is: $O(l + d^* \sum_{i=1}^{i=l} d_i^*)$, where l is the number of SACs, d_i^* is the degree of node i in the final augmented graph, and $d^* = \max_{i=1}^{i=l}\{d_i^*\}$. The relationship between d_i in our expression and d_i^* in their expression is this: $d_i \leq d_i^*$. Hence, the derivation provided here not only improves on the complexity bound but it is also expressed solely in terms of the computational graph.

4 The Hessian Algorithm with Preaccumulation

Preaccumulation has been introduced as a technique for reducing runtime as well as memory requirements in derivative computation in AD, especially in the first order case [18,19]. Here, we apply a similar statement-level preaccumulation strategy to the algorithm LIVARH described in the previous section. We show that preaccumulation can help reduce the overall cost of evaluating a Hessian. We first describe how the statement-level preaccumulation works and how it is incorporated to LIVARH to give LIVARHACC, and then we provide a complexity analysis of LIVARHACC.

4.1 Statement Level Preaccumulation

Consider an objective function given in the form of a piece of code, possibly containing branches. Even in the presence of branches, once the set of initial

values for independents is fixed, the evaluation of the objective function is exactly the same as the evaluation of assignment statements along the execution path. Here and in later sections we use the word 'statement' to refer to assignment statements to avoid confusion with SACs. Each statement defines a local scalar function, where the left-hand side variable represents the dependent variable, and all variables needed at the right-hand side are independent variables. Suppose we know the first and second order derivatives for each local function. Then, we can write a reverse mode Hessian algorithm that works on the statement level in exactly in the same way as we did in LIVARH. The only difference would be that in each step we process a local function which is defined by a statement rather than a SAC.

For the derivatives of each local function, notice that the evaluation of a statement is indeed equivalent to the evaluation of the SACs it corresponds to. So the first and second order derivatives for every local function can be computed by applying LIVARH on the SAC level. Thus, the algorithm with statement-level preaccumulation incorporated, LIVARHACC, divides the Hessian evaluation into two levels. In the first level, every SAC is processed to compute the first and second order derivatives for the local functions defined by statements. We refer to this as *local accumulation*. In the second level, the derivatives of local functions are processed to compute the derivatives for the objective function globally. We refer to this as *global accumulation*.

To successfully incorporate preaccumulation, first, we need to efficiently partition the SAC sequence based on statements (that is, group SACs according to statements). Because we only consider serial code here, we know that the SACs that belong to the same statement are contiguous in the SAC sequence. Second, we want to avoid roll-back reads during the processing. That is, we want to read and process each SAC only once. As we will show, both of these can be incorporated in our reverse mode framework at no additional cost.

In most codes, the last evaluated operator in an assignment statement is the basic assignment operator (=). In rather infrequent cases, other operators might come as the last operator; we will discuss how to deal with such cases in Section 5.3. For now let us assume that in our objective function every statement ends with the basic assignment operator. Let us further assume that there is no other assignment operator besides the last one in all statements. (This assumption is reasonable, since in operator overloading based AD tools implicit assignment of variables is an unsafe operation. It changes the precedence of operators, and people normally don't write codes containing that behavior.) Then, the assignment operators in the SAC sequences act as "splitters" for statements: they divide up the SAC sequence according to statements. If we read the SAC sequence in the reverse order, the "last" assignment operator will show up first.

So, to realize LIVARHACC, we simply maintain two sets of live variables, two sets of adjoints mapping, and two sets of Hessian mapping, in each case the one set dedicated for local accumulation and the other for global accumulation. Notationally, we have the sets (S_l, a_l, h_l) for local accumulation and

(S_g, a_g, h_g) for global accumulation. For each SAC, we work with (S_l, a_l, h_l) to compute the derivatives for a local function (statement). The updates during this phase are local because they only affect the results for each statement. When we encounter an assignment operator in the SAC sequence, we know that it is the beginning of a new statement and that we have finished computing the derivatives of the last statement—(S_l, a_l, h_l) contains the derivatives for the last local function. Before processing the new statement, we first do one global accumulation step; that is, apply LIVARH using (S_l, a_l, h_l) to update (S_g, a_g, h_g). The updates during this phase are global because they affect the final results. Then, we store the left-hand-side variable in the transient variable p and reset (reinitialize) (S_l, a_l, h_l) with respect to p for the next statement. After that, we process the assignment operator as the first SAC for a new local function. Figure 6 provides a high-level pseudocode for LIVARHACC as a whole, and Figure 7 provides an illustration of its workings.

4.2 Complexity Analysis

In this subsection we aim to answer the following question: Is incorporating statement-level preaccumulation in the Hessian algorithm beneficial? Just as we did in our analysis of LIVARH in Section 3.4, here again we assess complexity using the number of updates needed on the Hessian mapping. Recall that for LIVARHACC, we have to account for both local and global updates. The global updates take place only after all the SACs that belong to a statement have been processed. We want to compare the complexity of the two schemes LIVARH and LIVARHACC. To do so, instead of comparing the overall updates for the objective function, we focus on *one* statement and compare the number of local and global updates for LIVARHACC with the number of updates for LIVARH.

Suppose a statement is represented by k SACs, namely $\varphi_{i-k+1}, \cdots, \varphi_{i-1}, \varphi_i$ (this means v_i is the left hand side variable of the statement). Then, processing these k SACs would need $\sum_{j=0}^{k-1}(d_{i-j}+p_{i-j})(p_{i-j}+1) = U$ updates in LIVARH. In LIVARHACC, for each SAC, we perform local updates on h_l, and after all the k SACs have been locally processed, we perform global updates on h_g. Suppose the number of local updates is U_L, and the number of global updates is U_G. Then, assuming that every SAC is a unary or a binary operator, we can prove the following result (see the Appendix for proof):

Theorem 3 *Given a statement represented by k SACs with v_i as the left hand side variable, the number of local updates U_L and global updates U_G in* LIVARHACC *and the number of overall updates U in* LIVARH *are related by the inequality $U_L + U_G \leq U$ when $d_i(k-1) \geq 2k(k+1)$, where d_i is the number of non-linear interactions corresponding to v_i in* LIVARH.

It is interesting to note what Theorem 3 is saying for different values of k. When $k = 1$, which means preaccumulation degenerates to no preaccumulation, the condition in the statement cannot be satisfied, which is reasonable.

Algorithm: Enhanced Reverse Mode Hessian With Preaccumulation (LIVARHACC)
Input: a function f expressed as a SAC sequence $v_i = \varphi_i(v_j)_{v_j \prec v_i}, 1 \leq i \leq l$
Output: the global Gradient ∇f as $a_g(S_g)$, the global Hessian $\nabla^2 f$ as $h_g(S_g, S_g)$

Initialization:
$S_g = \{v_l\}, \quad a_g(v_l) = 1.0, \quad h_g(v_l, v_l) = 0.0, \quad S_l = \emptyset, \quad a_l = h_l = 0, \quad p = v_l$

for $i = l, \cdots, 1$ **do**

if $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ begins a new statement

% Update global live variables S_g
Set $w = a_g(p)$
Retrieve mapping $r : S_g \to \mathbb{R}$ as $r(v) = h(p, v)$
Remove p from S_g
for all $v \in S_l$ **do**
if $v \notin S_g$
Set $S_g = S_g \cup \{v\}, \quad a_g(v) = 0, \quad h_g(v, S_g) = 0$
% Update global adjoints $a_g(S_g)$
for all $a_l(v) \neq 0$ **do**
$a_g(v) = a_g(v) + a_l(v)w$
% Global Pushing
for all $v \neq p$ such that $r(v) \neq 0$ **do**
for all $u \neq v$ such that $a_l(u) \neq 0$
$h(v, u) += a_l(u)r(v)$
if $a_l(v) \neq 0$
$h(v, v) += 2a_l(v)r(v)$
if $r(p) \neq 0$
for all unordered pairs (v, u) such that $a_l(v)a_l(u) \neq 0$
$h_g(v, u) += a_l(v)a_l(u)r(p)$
% Global Creating
if $w \neq 0$
for all unordered pairs (v, u) where $h_l(v, u) \neq 0$ **do**
$h_g(v, u) += h_l(v, u)w$
% Initialize p and (S_l, a_l, h_l) for the next statement:
$p = v_i$
$S_l = \{v_i\}, \quad a_l(v_i) = 1, \quad h_l(v_i, v_i) = 0$
% Process $v_i = \varphi_i(v_j)_{v_j \prec v_i}$ locally
$\vdots$
(Same as LIVARH, just use S_l, a_l, h_l instead)

Fig. 6 The Hessian Algorithm with Preaccumulation (LIVARHACC).

Asymptotically, we only need $d_i > 2k$ for preaccumulation to pay off. The quantity k shows how complex an assignment in the source code is. We can expect k to be bounded because people don't normally write arbitrarily long statements in their codes. The quantity d_i, the number of nonzeros in the result variable's row in the Hessian matrix, indicates how sparse the Hessian matrix is. We can thus give the following intuitive explanation for why the condition suffices. In LIVARH, the cost for processing each SAC $\varphi_{i-j}, 0 \leq j < k$ is proportional to d_{i-j}, and $d_{i-j} \geq d_i$ (see the Appendix for proof). In LIVARHACC, the local cost is defined by k, which represents how complex the statement is, and

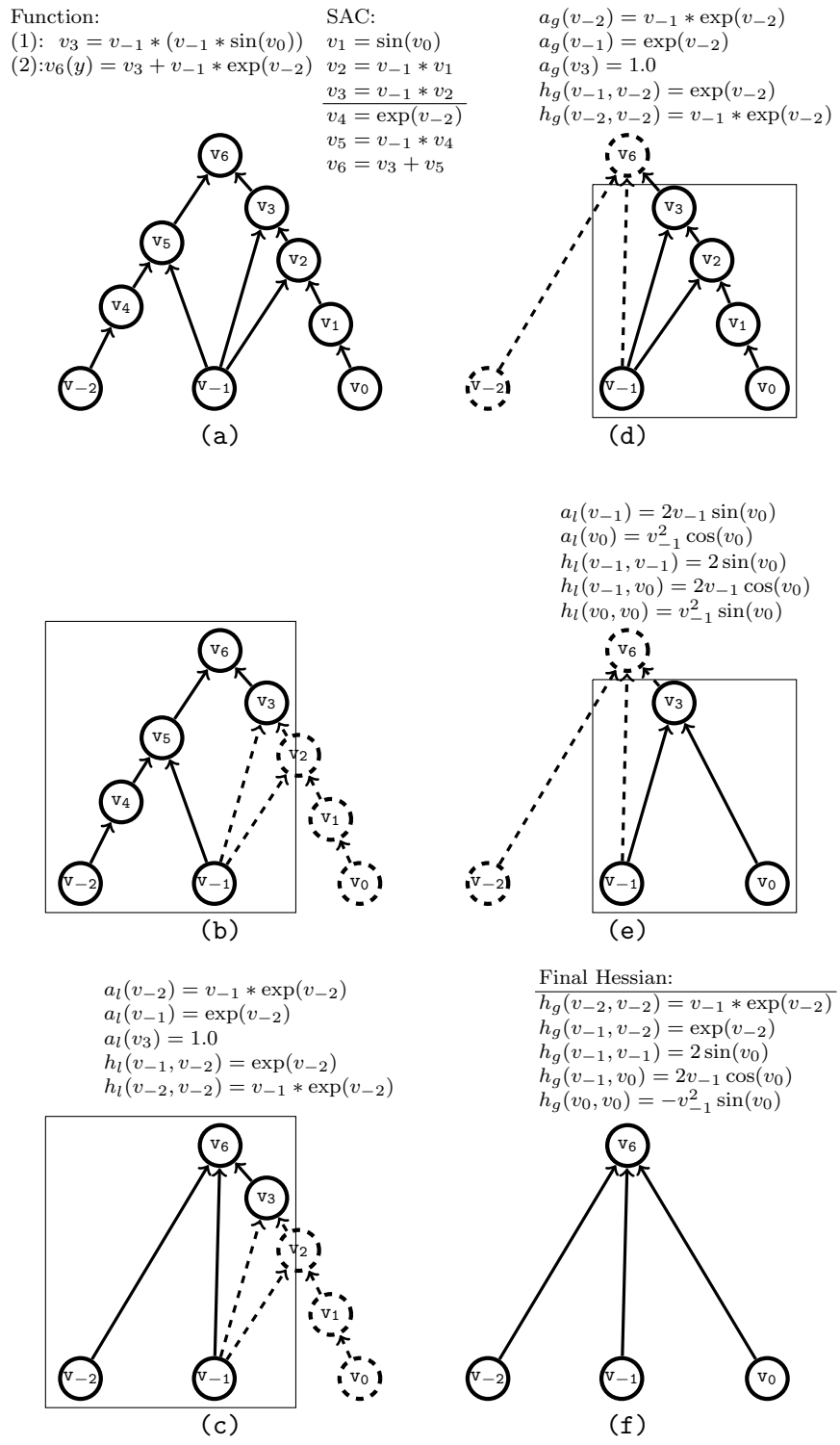


Fig. 7 An example illustrating how preaccumulation in the Hessian algorithm works. The example function consists of two statements, each decomposed into 3 SACs. Subfigure `(a)` gives the computational graph. Subfigure `(b)` shows local-accumulation for Statement (2). The rectangular box represents the sub-computational graph induced by Statement (2). After the local accumulation, we get the local results shown in subfigure `(c)`. Then, right before processing Statement (1), we need to performe one step of global accumulation, as depicted in subfigure `(d)`. The local accumulation of Statement (1) gives the results shown in subfigure `(e)`. Lastly, a global accumulation gives the final Hessian as shown in subfigure `(f)`.

the global cost is proportional to d_i. So, the benefit of preaccumulation comes from the fact that instead of letting each SAC to have a global interaction, we gather the derivatives locally and proceed globally at a coarser granularity. So for real problems in general, we expect the algorithm with preaccumulation to be superior.

Moreover, as will be discussed in the next section, the cost of each update depends on the size of the live variable set. This means that the local updates are cheaper than the global updates. So, even when the number of total updates in LIVARHACC is similar to that in LIVAR, we may still expect performance gain.

5 Implementation

We implemented both of the algorithms LIVARH and LIVARACC in ADOL-C. We discuss in this section details of the implementation including design decisions we made and the motivations behind them.

5.1 Index Translation

In native ADOL-C, a variable is represented by an integer index on the trace (called *tape* in ADOL-C). For each SAC, only the indices of the involved variables, the kind of operator and necessary constant(s) are stored. This scheme requires a forward sweep to prepare the intermediate values of every result variable v_i before a reverse sweep begins, because those values are needed when computing $\frac{\partial \varphi_i}{\partial v_j}$ in the reverse sweep.

In ADOL-C, the index of each variable is an integer representing a "memory location". Different variables may have the same index as long as their "live regions" do not overlap. If their regions overlap, then the indices are necessarily different. In particular, if S is a live variable set, then for every pair of distinct variables v_j, v_k in S, the index of v_j is different from the index of v_k. Variables without live-region overlaps, on the other hand, can use the same memory location. This means we cannot expect a guaranteed ordering on indices of variables. This phenomenon presents us with an issue in taking advantage of the symmetry available in the Hessian matrix/mapping in our algorithms.

We will explain why this is an issue in a moment, but first a note on notation. In our description of the algorithms so far, it appears as if we give each variable v_i an index i. This usage is only for clarity of presentation in the evaluation order of the SACs, but it is not necessarily mirrored in the implementation. To avoid potential misunderstanding, in what follows, we will use $l(v_i)$ to represent the index of a variable v_i in the implementation.

Since a Hessian is symmetric, we would prefer to store and work with only half of it in the Hessian mapping we use in our algorithms. Suppose we decide to store the lower half. Then, when we want to get all nonzeros of $r(l(v_i))$ for a SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$, what we actually do is get all nonzeros from the

entire row $l(v_i)$ of the Hessian matrix. Since we only store the lower half of the matrix, we have to search all rows below $l(v_i)$ to determine whether there is a variable v_j s.t. $h(l(v_j), l(v_i)) \neq 0$ and $l(v_j) > l(v_i)$. This entails a complexity proportional to the size of the current live variable set. But we would like to have a complexity proportional to only the number of nonzeros in that row.

The cause behind this undesirable higher complexity is, as alluded to earlier, the fact that the location index in ADOL-C does not guarantee an ordering. So, even if we are processing the SAC with result variable v_i using the location index $l(v_i)$, we cannot assume that all variables that have a non-linear interaction with v_i have smaller location indices than $l(v_i)$. If, in contrast, the indices of variables were monotonically increasing—that is, $i > j \Rightarrow l(v_i) > l(v_j)$—then, v_i would have had the largest index among S—that is, $l(v_i) = \max\{l(v_j) : v_j \in S\}$—since all v_j for $j > i$ would not yet have been evaluated at that point in time during the function evaluation. So, in such a scenario, we could safely get the nonzeros from just the lower half of the Hessian matrix for $l(v_i)$ as all nonzeros for the entire row.

Gower and Mello make this monotonic indexing assumption in their implementation, and they directly use the location indices provided by ADOL-C. The assumption, however, is not supported by ADOL-C, which is why their algorithm fails in some cases. In our implementation, we overcome this obstacle by translating the indexing scheme in ADOL-C into another, monotonically increasing indexing scheme. The routine we implemented for the translation is a simple and cheap operation that can be done in the forward sweep. In it, we recompute the objective function and store the results of each SAC for later use. We also keep an index counter and increase its value by one every time a SAC is processed. The index of each result variable v_i is assigned the value of the index counter. Finally, we also keep a mapping table so that future references of v_i as operands are redirected to the new index.

5.2 Implementing Livarh

In our implementations, we use the `map` container in the Standard Template Library (STL) in C++ as the main data structure to store the adjoint- and the Hessian-mappings. A typical implementation of `STL::map` uses a red-black tree, so search/insert/delete operations in `STL::map` each have $O(\log m)$ complexity, where m is the number of elements already in the data structure. Further, enumeration of elements is a constant time operation in `STL::map`. We implement the adjoint mappings as `map<int, double>` and the Hessian mappings as `map<int, map<int, double> >`.

With such data structures and the index translation discussed in Section 5.1 in place, we can implement Livarh in a relatively straightforward manner following the specification outlined in Figure 4. However in the implementation, we do not need to explicitly maintain the live variable set S since it is the *key set* of the adjoint mapping $a(S)$. Further, the temporary mapping $r(S)$ can be implemented using `map<int, double>`.

5.3 Implementing LIVARHACC

Implementing LIVARHACC (Figure 6) correctly and efficiently causes additional challenges beyond those encountered in the case of LIVARH. We discuss in this subsection three different aspects of these challenges (memory efficiency, complex statements, and optimization of the assignment operator in ADOL-C) and our approaches for addressing them.

5.3.1 Memory performance

The `STL::map` container is no longer a good solution for the *local* adjoints and Hessian mappings in terms of memory performance. In particular, since each statement is treated individually, an `STL::map` would need to frequently allocate and deallocate small chunks of memory during the local accumulation, entailing highly increased runtime. To circumvent this, instead of `STL::map`, we implement a memory-efficient customized data structure for just the local accumulation (the global accumulation still uses `STL:map`). The idea is to preallocate a fixed size memory, and work with simple linear searches (instead of the advanced data structure operations) to perform the updates. Since generally the number of SACs corresponding to a statement is bounded, the fixed memory would be of small size. In our implementation, we set the size to be large enough for 50 operators. (Since it is unlikely for a reasonable code to contain a statement with more than this many operators.) If in the code a statement with larger than 50 operators is encountered, memory is reallocated with double the size. So, this way, overall, we avoid the highly costly frequent memory allocation/deallocation that would occur had we used `STL:map` for the local accumulation by paying a much lower overhead for managing the fixed-size memory. In our experiments, we observed orders of magnitude performance improvement by following this strategy.

5.3.2 Complex statements

Implementing LIVARHACC also requires us to deal with special kinds of statements in programs. In the discussion in Section 4.1, we assumed that every statement in a piece of code ends with the basic assignment operator (=). Further, we used the assignment operator as the splitter to divide up a SAC sequence according to statements. Below we discuss how our implementation handles cases where these assumptions no longer hold.

Compound assignment operators. In `C++`, there are operators other than the basic assignment operator that can be the last operator in a statement. These are called *compound assignment operators*. A compound assignment operator consists of a binary operator and the basic assignment operator. The equivalent task is to perform the binary operation on the two operands and store the result on the left operand. For example, $a+ = b$ is equivalent to $a = a + b$. In implementing LIVARHACC, we treat compound assignment operators as

splitters in the SAC sequence. In addition to compound assignment operators, ADOL-C supports two other operators that combine a multiplication and an addition or a subtraction; these assignments are called `eq_prod_plus` ($+=*$) and `eq_prod_minus` ($-=*$). We treat these in the same way as we treat compound assignment operators. Table 2 gives a list of operators that work as splitters in the SAC sequence and their equivalent forms.

Table 2 Operators which serve as splitters in a SAC sequence. ADOL-C uses a suffix `_d` to distinguish between the cases where the right operand is a variable and where it is a constant. For `eq_plus_prod` and `eq_minus_prod` all operands must be variables.

Name (in `ADOL-C`)	Operator	Usage	Equivalent form
`assign_a, assign_d`	$=$	$a = b$	$a = b$
`eq_plus_a, eq_plus_d`	$+=$	$a+ = b$	$a = a + b$
`eq_minus_a, eq_minus_d`	$-=$	$a- = b$	$a = a - b$
`eq_mult_a, eq_mult_d`	$*=$	$a* = b$	$a = a * b$
`eq_plus_prod`	$+=*$	$a+ = b * c$	$a = a + b * c$
`eq_minus_prod`	$-=*$	$a- = b * c$	$a = a - b * c$

Trigonometric functions. Another issue we deal with involves trigonometric functions. In handling `sin/cos` operators, ADOL-C introduces implicit temporary variables to store the corresponding `cos/sin` values, to make it easier to compute the derivatives for the `sin/cos` operators. These temporary variable are initialized with the value zero, which entails the occurrence of an `assign_d` operator on the trace in the middle of the evaluation of a statement. This implicitly generated `assign_d` operator cannot be a splitter. But, without relevant context information, it is hard to know whether the `assign_d` operator comes from an explicit source function code or is introduced by ADOL-C implicitly.

Fortunately, there is a way to tell this apart. If the `assign_d` operator is indeed from the source function code, then the previous operator in the SAC sequence must also be a splitter operator. So, in our implementation, when an `assign_d` operator is encountered, instead of making a determination about it immediately, we keep a record of it and postpone its processing until we meet another operator that serves as a splitter. This approach won't affect the explicit `assign_d` operator as this is processed immediately. For the implicit ones, because they are introduced as temporaries, their processing is deferred as if they were explicitly written in the code before the assignment statement that generates them. Table 3 gives an example of the deferred `assign_d` operator.

5.3.3 Optimization of the basic assignment operator in ADOL-C

In ADOL-C, overloading of the basic assignment operator is optimized to reduce the storage requirement of the *tape*. In particular, if the right-hand-side of a statement is an expression, the last operator on the right hand side

Table 3 Example of a deferred `assign_d` operator. t_3 is the implicit temporary variable. When processing the SACs from the trace, the processing of $t_3 = 0$ is deferred until the next statement splitter is met. In effect we treat this as if the source code were written in the equivalent form shown in the right most column.

Source Code:	Trace (in `ADOL-C`):	Processing Order:	Equivalant to:
$x_1 += x_0$	$x_1 = x_1 + x_0$	$x_1 = x_1 + x_0$	$x_1 += x_0$
$y = \exp(x_0) * \sin(x_1)$	$t_1 = \exp(x_0)$	$\mathbf{t_3 = 0}$	$\mathbf{t_3 = 0}$
	$t_2 = \sin(x_1)$	$t_1 = \exp(x_0)$	$y = \exp(x_0) * \sin(x_1)$
	$\mathbf{t_3 = 0}$	$t_2 = \sin(x_1)$	
	$t_3 = \cos(x_1)$	$t_3 = \cos(x_1)$	
	$t_4 = t_2 * t_3$	$t_4 = t_2 * t_3$	
	$y = t_4$	$y = t_4$	

is made to "absorb" the basic assignment operator by directly assigning the result to the left-hand-side variable. For an illustration, consider Example 1. In this particular case, the information stored on the *tape* by ADOL-C in effect "by-passes" the temporary variable v_3, and the product $(v_{-1} * v_2)$ is directly assigned to y. In general, for every assignment statement other than simple copies such as $a = b$, ADOL-C avoids storage of information of the basic operator on the *tape*. This means with the current ADOL-C implementation, we cannot directly see any basic assignment operator on the SAC sequence stored on the *tape* unless the assignment is the copy statement $a = b$. Therefore, it is very difficult to isolate statements from the SAC sequence.

One way of solving this problem is for every operator to define an accompanying "assignment" version. That is, if the result of an operator is redirected to an assignee (which indicates that this operator is the last one on the right-hand-side and absorbs the basic assignment operator), we use the assignment version of the operator. Then when we encounter an assignment version of an operator in the reverse sweep, we know that another statement begins. There would be no increase in number of SACs associated with such an approach. However, implementing this requires rewriting a large portion of ADOL-C. Instead, in our current implementation of LIVARHACC, we simply *turn off* the basic assignment operator optimization. This leads to a longer SAC sequence for the same function evaluation—and thus, in theory, a higher complexity—compared to LIVARH where the optimization is left turned on. Nonetheless, we wanted to investigate how LIVARHACC performs even under such a setting.

6 Performance Evaluation

We present in this section experimental results aimed at evaluating the performance of the Hessian algorithms introduced in this paper in comparison with several existing Hessian algorithms. As testbeds, we use synthetic functions obtained from [20] and a mesh optimization problem obtained from the FeasNewt benchmark [21]. The set of results on the synthetic testbed is pre-

sented in Section 6.2 and the set of results on the mesh optimization testbed is presented in Section 6.3.

6.1 Algorithms Compared and Test Platform

A) Algorithms. We compare the performance of Livarh and Livarhacc against three existing SOAM-based approaches:

- `full-Hessian` (or `FullHess` for short): a full Hessian algorithm in which sparsity is *not* exploited at all,
- `sparseHess-direct` (or `Direct` for short): a compression-based direct sparse Hessian algorithm, and
- `sparseHess-indirect` (or `Indirect` for short): a compression-based indirect sparse Hessian algorithm.

The compression-based algorithms `Direct` and `Indirect` consist of four distinct steps [10]: sparsity pattern detection ($S1$), computation of *seed* matrix via graph coloring ($S2$), computation of the compressed Hessian matrix-seed matrix product ($S3$), and recovery of the original Hessian entries from the compressed representation ($S4$). The underlying methods for the coloring and recovery steps in both `Direct` and `Indirect` (developed in our earlier work [9, 10]) are implemented in the package `ColPack` [22] and coupled with ADOL-C for seamless usage. We access these via the `sparse_hess()` drivers in ADOL-C, which in turn invoke `ColPack` routines for the coloring and recovery steps.

The difference between `Direct` and `Indirect` lies in how the compression-recovery trade-off is handled: in `Direct` the compression is done such that the recovery is achieved in a direct fashion whereas in `Indirect` a tighter compression that requires recovery via substitution is employed. The graph coloring model underlying `Direct` is *star coloring* and the model behind `Indirect` is *acyclic coloring* [9,10]. Acyclic coloring uses fewer colors than star coloring, which leads to a lower complexity in computation of the compressed Hessian ($S3$). Meanwhile, the coloring step ($S2$) as well as the recovery step ($S4$) in `Indirect` are of higher complexity than the respective steps in `Direct`. For sparsity detection (in both methods), we use the recent forward mode-based algorithm described in [24] whose implementation is available in the `ADOL-C` version we used. (Older versions of ADOL-C use an earlier sparsity detection algorithm presented in [23].)

We could not include results on Gower and Mello's implementation of the Edge Pushing algorithm in our comparison, since as mentioned previously, their code gives incorrect results in most of the test cases. (We downloaded the Edge Pushing code (on October 6th, 2014) from the author's webpage at
`http://www.maths.ed.ac.uk/~s1065527/software.html`
and had difficulties to make it work correctly. As an example, the routine which evaluates the Hessian computes incorrect results so subsequent routines which convert the results into sparse format either give incorrect results or cause a segmentation fault.) If we run the first routine despite the incorrect results, the runtime is within a factor of two of that of our Livarh implementation.

B) Platform. For all of the results we report in the paper, the experiments are conducted on a computer equipped with a Quad Core 2.5 GHz Intel I5-2400S processor, 8 GB Memory. The codes are compiled using `gcc/g++` 4.8.2 with optimization flag `-O3` and we use ADOL-C version 2.5.2. We implemented LIVARH and LIVARHACC on top of ADOL-C. The only change we made to native ADOL-C—besides of course the new implementations—is that when running LIVARHACC, we disable the basic assignment operator optimization as described in Section 5.3.3.

C) Runtimes. For each test we perform, we report the arithmetic mean of 20 runs and the coefficients of variation (standard deviation divided by the mean, CV) for the runs. We exclude outliers which are results three standard deviations away from the mean because they are most likely due to an unstable status of the system (routine background jobs, etc). We observed only two outliers in all our experiments.

D) Source Code. We have made our implementations publicly available as open-source at `https://github.com/CSCsw/LivarH`. Similarly, we have made the testbeds available at `https://github.com/CSCsw/LivarH-Test`.

6.2 Synthetic Functions, Regular Structures

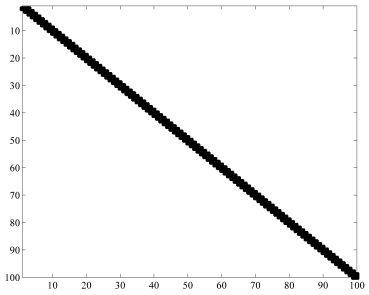
A) Test Functions and Setup. For our tests using synthetic functions, we hand-picked four functions from the collection in [20]. We chose the functions such that the Hessians would represent varying sparsity structures. We hand-coded the functions for the purposes of applying AD. In all the tests, we set the number of independent variables to be $20,000$; each Hessian is therefore a $20,000 \times 20,000$ matrix. Table 4 lists the names of the four test functions, the number of nonzeros (nnz) in the lower half (including the diagonal) of each Hessian, and the average number of nonzeros per row in each Hessian. (All of the algorithms we compare except for `Full-Hessian` return only the upper or lower half of the Hessian matrix.) We plot in Figure 8 the sparsity structure of the Hessians of the four functions; to ease visualization, we use smaller-size problems (100×100) in these plots rather than the actual dimension used in the Hessian computations ($20,000 \times 20,000$). In the Appendix A.3, we list the mathematical expressions specifying the four test functions.

Table 4 List of the synthetic test functions. In each case, number of rows $n = 20,000$.

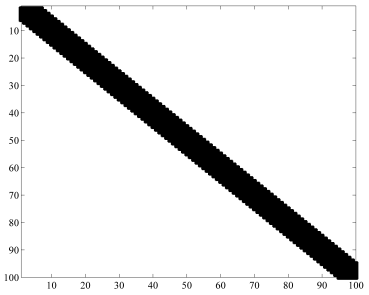
Func	Name	nnz in lower H	avg $\lceil nnz/row \rceil$
`F1`	Chained Rosenbrock function	$39,999$	3
`F2`	Generalized Broyden banded function	$119,985$	11
`F3`	Potra and Rheinboldt boundary value problem	$89,997$	7
`F4`	Gomez-Ruggiero function	$159,972$	8

Table 5 Properties of the synthetic functions when decomposed into single assignment codes. Numbers are shown with the basic assignment operator optimization in `ADOL-C` turned on or off.

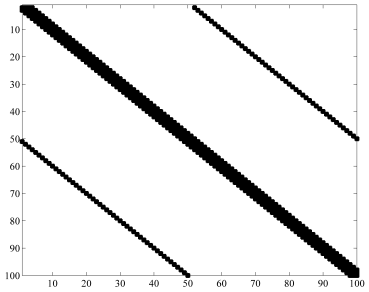
Func	No. SACs (assign. optimization on)	No. SACs (assign. optimization off)	No. Statements
`F1`	179,998	179,998	20,000
`F2`	459,964	579,949	139,986
`F3`	259,989	279,987	39,998
`F4`	240,008	260,009	80,000



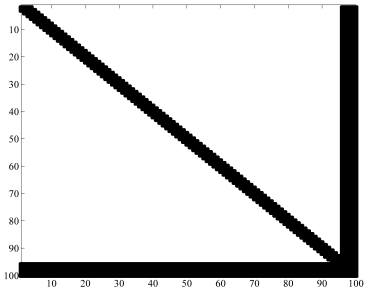
(a) F1



(b) F2



(a) F3



(b) F4

Fig. 8 Sparsity structure of the Hessians of the synthetic test functions.

Table 5 lists the properties of the synthetic functions when decomposed into single assignment codes. The second column gives the number of SACs recorded on ADOL-C *tape* with the basic assignment operator optimization turned on. The third column is the number of SACs recorded on ADOL-C *tape* when we run LIVARHACC, i.e. the basic assignment operator optimization turned off. The last column lists the number of statements in each test function. The number of SACs tells us about the numerical complexity of the function. The number of statements gives us an idea about how the func-

tions are implemented. For example, `F1` and `F3` are implemented with long statements, whereas `F2` and `F4` are implemented with short statements.

B) Runtime results. Table 6 shows the performance results (in terms of overall runtime in seconds) of the various algorithms on these test functions. Table 7 shows the runtime breakdown for the `Direct` and `Indirect` methods as well as the number of colors needed in the underlying `star` and `acyclic` colorings for each test function. We comment on a few points on the results seen in these tables. The `Direct` and `Indirect` methods have been experimentally evaluated over several test problems earlier [9,10,22]. And hence, here we focus on the new live variable algorithms.

Table 6 Average runtimes (in sec) of 20 runs of the different Hessian algorithms on the synthetic test functions and the corresponding coefficients of variation (CV) percentage (in parentheses).

Func	`FullHess`	`Direct`	`Indirect`	LIVARH	LIVARHACC
`F1`	126.8 (1.03%)	0.072 (0.58%)	0.851 (0.13%)	0.094 (3.95%)	**0.043** (0.96%)
`F2`	1723.1 (3.96%)	1.002 (2.92%)	2.353 (5.06%)	0.539 (1.50%)	**0.224** (2.44%)
`F3`	145.4 (1.14%)	0.622 (13.7%)	1.546 (0.38%)	0.260 (1.05%)	**0.090** (0.66%)
`F4`	115.4 (0.13%)	49.89 (16.1%)	106.9 (7.90%)	0.259 (0.61%)	**0.130** (0.81%)

Table 7 Breakdown of runtimes (in sec) and number of colors needed for `Direct` and `Indirect`. The CV values are shown in parentheses.

Func	`Direct (star)`				
	No. colors	$S1$	$S2$	$S3$	$S4$
`F1`	4	0.027 (0.79%)	0.015 (1.41%)	0.027 (1.06%)	0.003 (0.93%)
`F2`	11	0.344 (0.97%)	0.078 (0.47%)	0.573 (5.00%)	0.007 (3.95%)
`F3`	9	0.509 (0.54%)	0.056 (1.60%)	0.069 (0.83%)	0.006 (4.79%)
`F4`	10	49.77 (16.1%)	0.066 (1.41%)	0.049 (0.50%)	0.006 (5.15%)
Func	`Indirect (acyclic)`				
	No. colors	$S1$	$S2$	$S3$	$S4$
`F1`	2	0.028 (0.70%)	0.014 (0.62%)	0.014 (0.71%)	0.796 (0.12%)
`F2`	6	0.343 (0.75%)	0.045 (0.74%)	0.294 (4.13%)	1.671 (7.29%)
`F3`	6	0.510 (0.63%)	0.038 (1.04%)	0.046 (1.29%)	0.953 (0.48%)
`F4`	8	53.59 (15.4%)	52.41 (2.12%)	0.037 (0.31%)	0.858 (22.0%)

First, the results show that computing Hessians disregarding sparsity is wasteful and for large problem sizes even infeasible. Second, in all test functions except for `F1`, the new algorithm LIVARH is found to be faster than both of the compression-based methods. The function `F1` is an exception because of the extremely simple structure of its Hessian: a tridiagonal matrix, which can be computed using two or three Hessian-vector products. Third, in all cases, LIVARHACC is the fastest method (in bold) and on these problems it is

around two to three times faster than LIVARH. Fourth, focusing just on the compression-based methods, we can see that the `Direct` (star coloring-based) method is faster than the `Indirect` (acyclic coloring-based) method for the test functions considered. The reason for this is the banded structure of the matrices: the cost associated with recovery for the indirect method outweighs the reduction in number of colors afforded by acyclic coloring relative to star coloring, making the direct method faster overall. Finally, we note that for `F4`, whose Hessian is an arrow-head matrix, both `Direct` and `Indirect` methods take extraordinarily long times. This is primarily because (as can be seen in Table 7) the sparsity detection step takes exceptionally long time for this particular structure. Also, the breakdown results show that the sparsity detection step and the recovery step for `Indirect` method have high coefficient of variation, which explains the high coefficients of variation of the overall timing results for `Direct` and `Indirect` methods on `F4`. In contrast, the results show that the LIVARH algorithm and the LIVARHACC algorithm are more robust to variations in sparsity structure.

C) Special sparsity structures. Regarding `F4`, we also see that acyclic coloring ($S2$ in `Indirect`) takes extraordinarily long time. This is due to the special nature of this structure vis-a-vis the intricacies of the acyclic coloring algorithm and its implementation. For the purpose of being general, the implementation of acyclic coloring involves maintaining a disjoint-set data structure that keeps track of two-colored induced subgraphs (trees) [9]. The arrow-head matrix structure and the associated adjacency graph make the acyclic coloring algorithm's search and merge operations on this data structure especially intensive, which explains the large runtime needed. One can offer a similar explanation for the high runtime cost of the sparsity detection routine for this same testcase.

However, for this case, the sparsity pattern of the Hessian can in fact be directly inferred from the expression of the function—thus entirely avoiding having to call the sparsity detection routine in ADOL-C. Similarly, an acyclic coloring of an arrow-head matrix can also be trivially obtained without having to call ColPack—just assign each column/row in the dense column/row portion of the matrix a distinct color, and then assign all remaining columns/rows a new color. This would in fact give an optimal acyclic coloring (and the acyclic coloring results in Table 7 output by ColPack do exactly match this). If the sparsity detection and coloring for the function `F4` were done as just described, then the overall runtime for the compression-based methods would essentially be $S3$ plus $S4$, which would give about 0.89 seconds for `Indirect` and 0.06 seconds for `Direct`, runtimes that are a lot closer to those of LIVARH and LIVARHACC. The lesson therefore is that one should avoid using generic sparsity pattern detection and coloring routines for special cases, such as the arrow-head structure, for which solutions can be trivially obtained.

D) Repeated Hessian computations. In certain iterative optimization procedures, one may need to repeatedly compute Hessians for a given input function.

In such cases, the sparsity structure detection and the graph coloring steps of the compression-based approach for sparse Hessian computation need to be performed only once, since the sparsity pattern remains the same throughout the iterations. Under such circumstances, the compression-based methods (which would then have runtime equal to just $S3$ plus $S4$) could be faster than the LIVARH algorithm. In our synthetic test functions, we see this to be the case in all functions except for `F3`, where the runtime needed for only computing the compressed Hessian and then recovery ($S3$ plus $S4$) using the `Direct` method is less than the runtime of LIVARH or LIVARHACC. For repeated Hessian computation when the sparsity pattern remains the same, the compression-based approach should be preferred over LIVARH or LIVARHACC.

E) Memory usage. Another metric besides runtime we use to compare the various algorithms is memory footprint. Using `Valgrind 3.10.1` [25], we measured the heap memory usage of the different algorithms. Table 8 gives the results obtained on these same synthetic test functions. We omit the memory usage for the full Hessian algorithm because it requires us to pre-allocate the whole Hessian matrix in memory, in this case a $20,000 \times 20,000$ matrix, which would need at least 1.6GBytes. Clearly this is inefficient and infeasible for larger cases. From the result, first we see that all the four algorithms use only 20 to 50 Mbytes of memory. And for same function, the four algorithms use roughly the same amount of memory. `FullHess` algorithm requires at least 1.6 GBytes of memory for these problems and this is another reason that sparsity should always be considered when the Hessian to be evaluated is sparse. Second, the `Indirect` algorithm uses more memory compared to the `Direct` algorithm (expect for `F2`). This is due to the former using the acyclic coloring algorithm with the additional book-keeping information this coloring entails, whereas the `Direct` algorithm uses star coloring which requires maintaining less information [9]. Finally, the extra memory usage of LIVARHACC compared to LIVARH can be explained by Table 5. As can be seen there, our implementation of LIVARHACC works on a longer SAC sequence, and therefore needs more memory.

Table 8 Memory usage (in MB) of the different Hessian algorithms on the synthetic test functions.

Func	`Direct` (star)	`Indirect` (acyclic)	LIVARH	LIVARHACC
`F1`	19.63	25.70	25.09	25.13
`F2`	49.94	49.94	48.07	54.48
`F3`	27.27	40.56	33.37	34.47
`F4`	27.36	45.07	33.65	34.75

F) Update count. In our analyses in Sections 3.4 and 4.2, we used the number of updates on the Hessian mapping as a means to gauge the complexity of the

algorithms LIVARH and LIVARHACC. Table 9 shows the *observed* number of Hessian-updates these algorithms performed during their entire execution for each of the test functions. We can see a substantial reduction in total number of updates due to preaccumulation in all testcases except for `F1`. The reduction fairly accurately translated to the observed performance (runtime) gain for LIVARHACC over LIVARH. Here, `F1` is an exception because of its extreme simplicity. In its implementation, each loop just contains one line of code which sums up a local expression. So the global accumulation degenerates to simply copying things again. In this case, the performance gain of LIVARHACC comes from the fact that performing local updates is much more efficient than global updates.

Table 9 Number of updates performed in LIVARH and LIVARHACC on the synthetic test functions.

Func	LIVARH	LIVARHACC		
	Total Ops	Local Ops	Global Ops	Total Ops
`F1`	199,990	199,990	59,997	259,987
`F2`	3,219,515	339,970	959,845	1,299,815
`F3`	1,279,872	49,995	249,975	299,970
`F4`	1,199,978	60,000	399,992	459,992

6.3 Mesh Optimization, Irregular Structure

A) Problem setup. We did an evaluation and analysis similar to what we have done with the synthetic test functions in Section 6.2 on a mesh optimization problem as an example of a real-world problem. The mesh optimization problem and the function code are described in [21]. We have, however, made some small changes in using the code. Specifically, in the original code, there are fixed nodes and non-fixed nodes, and the code only computes the Hessian for the non-fixed nodes. To simplify the problem, here we treat all nodes as non-fixed and compute the Hessian via AD through the objective function directly.

In our experiments using this modified mesh optimization code, we consider three meshes with different sizes: `gear`, `duct12` and `duct8`. The number of independent variables n for them is equal to $2,598$, $11,597$ and $39,579$ respectively. The lower half of the three corresponding Hessian matrices contain number of nonzero entries (nnz) equal to $46,488$; $253,029$ and $828,129$, respectively. Table 10 summarizes these numbers along with properties of the decompositions of the problems into SACs, again with the basic assignment operator optimization in `ADOL-C` turned on or off. We can see that these Hessians are denser than the Hessians of the synthetic functions we considered in Section 6.2, but they are still sufficiently sparse (compare nnz with n^2 in each case). In Figure 9 we give a plot of the sparsity pattern of the Hessian matrix

of the smallest problem, `gear` where $n = 2,598$. In contrast to the Hessians of the synthetic function we saw earlier, we can see that the sparsity pattern of these Hessians is irregular.

Table 10 Structural and runtime properties of the mesh optimization problems.

Mesh	n	nnz in lower H	No. SACs (assign. opt. on)	No. SACs (assign. opt. off)	No. Statements
`gear`	2,598	46,488	242,563	289,303	87,249
`duct12`	11,597	253,029	1,492,730	1,781,061	538,217
`duct8`	39,579	828,129	5,088,836	6,072,449	1,836,073

Fig. 9 Sparsity structure of the Hessian of the smallest mesh problem (`gear with` $n = 2,598$).

B) Runtime results. Table 11 shows the overall runtime (in seconds) results of the various Hessian algorithms on these problems, and Table 12 shows the runtime breakdown for `Direct` and `Indirect`. A few comments are in order. First, we can see that LIVARH is around six to eight times faster than both `Direct` and `Indirect`, and LIVARHACC gives further performance improvement, in most cases more than 30% over LIVARH. Second, unlike the results we saw in the synthetic case, here the algorithm `Indirect` (based on acyclic coloring) is found to be comparable to `Direct` (based on star coloring). This is because of the irregularity of the sparsity pattern and the associated difference in colors versus recovery-cost tradeoff. Finally, we see that sparsity pattern detection is the costliest step (of the four steps in a compression-based method)

Table 11 Average runtimes (in sec) of 20 runs of the different Hessian algorithms on the mesh optimization problems. The corresponding CV values of the 20 runs are shown in parentheses.

Mesh	`FullHess`	`Direct`	`Indirect`	LIVARH	LIVARHACC
`gear`	18.96 (0.39%)	3.360 (1.01%)	3.280 (1.41%)	0.520 (0.36%)	**0.326** (0.92%)
`duct12`	>1 hour	34.47 (2.41%)	28.49 (1.38%)	3.548 (1.34%)	**2.230** (1.33%)
`duct8`	>2 hours	119.3 (1.27%)	108.5 (1.99%)	12.75 (1.43%)	**7.960** (1.48%)

Table 12 Breakdown of runtimes (in sec) and number of colors needed on the mesh problems. The CV values are shown in parentheses.

Mesh	`Indirect (acyclic)`				
	No. colors	$S1$	$S2$	$S3$	$S4$
`gear`	54	2.897 (1.16%)	0.064 (0.43%)	0.396 (0.43%)	0.002 (2.51%)
`duct12`	62	19.40 (2.21%)	0.396 (0.30%)	14.66 (4.54%)	0.012 (3.59%)
`duct8`	65	69.02 (1.69%)	1.350 (1.02%)	48.86 (1.40%)	0.049 (4.26%)
Mesh	`Indirect (acyclic)`				
	No. colors	$S1$	$S2$	$S3$	$S4$
`gear`	31	2.893 (1.63%)	0.049 (2.34%)	0.232 (1.42%)	0.106 (0.69%)
`duct12`	30	19.57 (1.86%)	0.302 (0.67%)	6.993 (2.39%)	1.627 (0.18%)
`duct8`	31	69.11 (1.90%)	1.094 (0.56%)	23.69 (8.06%)	14.59 (0.20%)

in both of these algorithms. A symbolic live variable approach could be used to speed up this step, and this is the scope of future work.

C) Memory usage. Table 13 lists results on memory usage of the various algorithms on these problems. The results show that LIVARH uses less memory than both `Direct` and `Indirect`. The results also show that LIVARHACC requires more memory than LIVARH, again because more SACs are stored for LIVARHACC. For example, for the largest problem (mesh `duct8` with $n = 39,579$), we can see from Table 10 that LIVARHACC works with over one million more SACs than the other algorithms.

D) Updates count. Table 14 gives the number of observed updates on the Hessian mappings involved in LIVARH and LIVARHACC. In all cases, we can see that LIVARHACC reduces the total number of operations by a third over LIVARH. This, again, corroborates with the performance results shown in Table 11.

Table 13 Memory usage (in MB) of the different Hessian algorithms on the mesh problems.

Mesh	`Direct` (star)	`Indirect` (acyclic)	LIVARH	LIVARHACC
`gear`	67.1	67.1	27.7	30.2
`duct12`	380.5	380.5	135.3	150.8
`duct8`	1253.0	1253.0	444.4	497.1

Table 14 Number of updates performed in LIVARH and LIVARHACC for the mesh problems.

Mesh	LIVARH	LIVARHACC		
	Total Ops	Local Ops	Global Ops	Total Ops
`gear`	3,134,934	136,658	1,917,115	2,053,773
`duct12`	19,558,132	845,318	12,018,204	12,863,522
`duct8`	66,631,857	2,881,914	40,949,715	43,831,629

7 Conclusions

We showed that Hessians in AD can and should be computed *directly* using the reverse mode. Recognizing the role of live variables in incremental reverse mode AD, we identified a key invariant that serves as a foundation for the proposed framework for Hessian algorithms. We also showed that preaccumulation is beneficial in this framework.

We implemented the Hessian algorithms we developed, LIVARH and LIVARHACC, within the operator overloading paradigm. The reader can see that the algorithms "map" naturally to an operator overloading implementation. There is, meanwhile, nothing inherent in the algorithms that precludes an implementation (of appropriate variants of the algorithms) within the source code transformation paradigm as well. The required adaptation, however, is likely to present a formidable challenge.

There are several worthwhile directions for extending this work. We mention examples of avenues we intend to pursue.

- The key invariants we observed in this work can be naturally extended to orders higher than two. For example, extending the invariant to third order would mean maintaining a mapping $\mathcal{T}$ from every unordered triplet (u, v, w) in S to a real value $\mathcal{T}(u, v, w)$. The third order derivative tensor can thus be represented by the mapping $\mathcal{T}$ after each step. R. M. Gower and A. L. Gower [13] have recently extended their Edge Pushing algorithm to third-order. Their approach gives the directional derivatives for the Hessian, which is a slice of the third-order tensor, and hence it is similar to a forward-over-reverse mode which gives the directional derivatives for the adjoints.
- As we have discussed, the sparsity pattern detection step in the compression-based approach has the highest memory usage. A "symbolic" variant of LIVARH which merely computes the sparsity pattern of the Hessian would use less memory than the numeric LIVARH. We will consider using a symbolic version of LIVARH as the sparsity pattern routine in our compression-based methods; Gower and Mello[12] have demonstrated that a symbolic version of their Edge Pushing algorithm is suitable for compression-based methods. Compression-based methods are advantageous for cases where a Hessian matrix with the same sparsity pattern needs to be repeatedly evaluated within an iterative optimization procedure (or nonlinear equation solver), or when it is preferable to compute Hessian-vector products rather than the full Hessian at once.

– It would be interesting to find ways in which LIVARH and LIVARHACC can be parallelized on multicore and manycore architectures.

Acknowledgements

We thank Jean Utke, Andrea Walther and Kshitij Kulshreshta for their comments on earlier versions of this manuscript and discussions around ADOL-C. We thank Paul Hovland and Todd Munson for making the FeasNewt benchmark available to us. This work was supported by the U.S. National Science Foundation grants CCF-1218916 CCF-1552323, and the U.S. Department of Energy grant DE-SC0010205.

A Appendix

We provide here details that we left out from the discussions in Sections 2 thru 6. In Section A.1 we give an alternative, more abstract derivation of the Equations (6) and (8) from Section 3 on adjoints and Hessians, respectively. In Section A.2 we analyze the complexity of processing one statement in LIVARH and LIVARHACC, and then we establish a sufficient condition under which LIVARHACC necessarily reduces the total number of updates needed. In Section A.3 we give a listing of the mathematical expressions describing the test functions `F1`, `F2`, `F3` and `F4` used in Section 6.2.

A.1 Derivation of The Hessian Invariant Equation

Suppose we are about to process a SAC $v_i = \varphi_i(v_j)_{v_j \prec v_i}$. Following the notations from Section 2, the current live variable set is S_{i+1} and the objective function is equivalent to $\mathbf{F_{i+1}}(S_{i+1})$. After the SAC is processed, the live variable set will be S_i and the objective function will be equivalent to $\mathbf{F_i}(S_i)$. The relation between $\mathbf{F_{i+1}}(S_{i+1})$ and $\mathbf{F_i}(S_i)$ is that v_i is considered an independent variable in $\mathbf{F_{i+1}}(S_{i+1})$ whereas it is viewed as the implicit function $v_i = \varphi(v_j)_{v_j \prec v_i}$ in $\mathbf{F_i}(S_i)$. Thus:

$$\begin{aligned}\mathbf{F_{i+1}}(S_{i+1}) &= \mathbf{F_{i+1}}(S_{i+1} \setminus \{v_i\}, v_i) \\ &= \mathbf{F_{i+1}}(S_{i+1} \setminus \{v_i\}, v_i = \varphi_i(v_j)_{v_j \prec v_i}) \\ &= \mathbf{F_i}(S_{i+1} \setminus \{v_i\} \cup \{v_j | v_j \prec v_i\}) \\ &= \mathbf{F_i}(S_i).\end{aligned}$$

Let us introduce a notational short-hand at this point. Namely, to distinguish the partial derivative operation applied on $\mathbf{F_{i+1}}(S_{i+1})$ from that applied on $\mathbf{F_i}(S_i)$, we use $\frac{\partial}{\partial v}$ to denote the operation on $\mathbf{F_{i+1}}(S_{i+1})$ and $\frac{\hat{\partial}}{\hat{\partial} v}$ to denote the operation on $\mathbf{F_i}(S_i)$. Then we have:

$$\begin{aligned}\frac{\hat{\partial}\mathbf{F_i}(S_i)}{\hat{\partial} v} &= \frac{\hat{\partial}\mathbf{F_i}(S_{i+1} \setminus \{v_i\} \cup \{v_j | v_j \prec v_i\})}{\hat{\partial} v} \\ &= \frac{\partial \mathbf{F_{i+1}}(S_{i+1} \setminus \{v_i\}, v_i = \varphi_i(v_j)_{v_j \prec v_i})}{\partial v} \\ &= \frac{\partial \mathbf{F_{i+1}}(S_{i+1})}{\partial v} + \frac{\partial \varphi_i}{\partial v} \frac{\partial \mathbf{F_{i+1}}(S_{i+1})}{\partial v_i}.\end{aligned}$$

This is the same as the adjoints Equation (6). Viewed in terms of operators, the relationship is:

$$\frac{\hat{\partial}}{\hat{\partial} v} = \frac{\partial}{\partial v} + \frac{\partial \varphi_i}{\partial v} \frac{\partial}{\partial v_i}.$$

Now we wish to extend this relation to second order. To simplify notation, let $f = \mathbf{F_{i+1}}(S_{i+1})$ and $\hat{f} = \mathbf{F_i}(S_i)$. Then by applying the first order operator relation twice, we have:

$$\begin{aligned}
\frac{\hat{\partial}^2 \hat{f}}{\hat{\partial} v_j \hat{\partial} v_k} &= \frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\hat{\partial} \hat{f}}{\hat{\partial} v_k}\Big] = \frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial f}{\partial v_k} + \frac{\partial \varphi_i}{\partial v_k}\frac{\partial f}{\partial v_i}\Big] \\
&= \frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial f}{\partial v_k}\Big] + \frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial \varphi_i}{\partial v_k}\frac{\partial f}{\partial v_i}\Big] \\
&= \frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial f}{\partial v_k}\Big] + \frac{\partial \varphi_i}{\partial v_k}\frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial f}{\partial v_i}\Big] + \frac{\partial f}{\partial v_i}\frac{\hat{\partial}}{\hat{\partial} v_j}\Big[\frac{\partial \varphi_i}{\partial v_k}\Big] \\
&= \Big[\frac{\partial}{\partial v_j} + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial}{\partial v_i}\Big]\frac{\partial f}{\partial v_k} + \frac{\partial \varphi_i}{\partial v_k}\Big[\frac{\partial}{\partial v_j} + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial}{\partial v_i}\Big]\Big[\frac{\partial f}{\partial v_i}\Big] \\
&+ \frac{\partial f}{\partial v_i}\Big[\frac{\partial}{\partial v_j} + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial}{\partial v_i}\Big]\Big[\frac{\partial \varphi_i}{\partial v_k}\Big] \\
&= \frac{\partial^2 f}{\partial v_j \partial v_k} + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial^2 f}{\partial v_i \partial v_k} + \frac{\partial \varphi_i}{\partial v_k}\frac{\partial^2 f}{\partial v_j \partial v_i} + \frac{\partial \varphi_i}{\partial v_k}\frac{\partial \varphi_i}{\partial v_j}\frac{\partial^2 f}{\partial v_i \partial v_i} + \frac{\partial f}{\partial v_i}\frac{\partial^2 \varphi_i}{\partial v_j \partial v_k} \\
&= \frac{\partial^2 f}{\partial v_j \partial v_k} \\
&+ \frac{\partial \varphi_i}{\partial v_j}\frac{\partial^2 f}{\partial v_i \partial v_k} + \frac{\partial \varphi_i}{\partial v_k}\frac{\partial^2 f}{\partial v_j \partial v_i} + \frac{\partial \varphi_i}{\partial v_j}\frac{\partial \varphi_i}{\partial v_k}\frac{\partial^2 f}{\partial v_i \partial v_i} + \frac{\partial^2 \varphi_i}{\partial v_j \partial v_k}\frac{\partial f}{\partial v_i}.
\end{aligned}$$

This is an alternate, from first-principles derivation of the Hessian Equation (8).

A.2 Analysis of LIVARHACC

We follow the assumption in Section 4.2 that every operator is either unary or binary. Suppose a statement is represented by k SACs, $\varphi_{i-k+1}, \cdots, \varphi_{i-1}, \varphi_i$, meaning v_i is the left hand side variable of the statement. This statement defines a local computational graph that is embedded in the global computational graph. (See Figure 7 (b) and (d) for an illustration.) The embedded local computational graph has the following properties:

(1) The local computational graph is almost a tree. The root node of the tree is v_i, the "leaf" nodes are variables that appear in the right hand side of the statement, and the intermediate nodes are $v_{i-k+1}, \cdots, v_{i-1}$. Only the root node and the "leaf" nodes are explicitly declared variables in the code. The intermediate nodes $v_{i-k+1}, \cdots, v_{i-1}$ are implicitly generated by the complier at run-time.
(2) The local computational graph differs from a tree in that only the "leaf" nodes in the local computational graph can have out-degree more than one (the local subgraph looks like an inverted funnel).

These two properties hold true primarily because of two basic assumptions we make. First, we assume that there is no implicit assignment in AD, since it is not safe. The evaluation order might be different from what is expected for intrinsic floating-point types. Second, for overloaded-operators, a compiler does not eliminate common sub-expressions (or at least cannot safely do it for free).

From the aforementioned two properties, the following claim follows.

Lemma 1 *Using the notations and assumptions established earlier, in* LIVARH*, let d_{i-j} denote the number of nonzeros in the row v_{i-j} when processing $\varphi_{i-j}, 0 \leq j < k$. Then, $d_i \leq d_{i-j}$.*

Proof At the beginning of processing φ_i, the other ends of the nonlinear interaction are explicit variables. For all φ_{i-j}, $0 < j < k$, the result variable v_{i-j} is an implicit variable, as stated in Property (1). Therefore, there will be no merge of nonlinear interactions. In other words, the variable v_{i-j} gets all nonzero entries from its parent, as illustrated in Figure 7 (a) and (b). So the statement follows immediately. □

From the discussion in Section 4.2, we know that in LIVARH, an upper bound on the total number of updates needed in the Hessian mapping to process the SACs $\varphi_{i-k+1}, \cdots, \varphi_{i-1}, \varphi_i$ can be given by:

$$U = \sum_{j=0}^{k-1} \Big(d_{i-j} p_{i-j} + p_{i-j}(p_{i-j} + 1) \Big). \tag{13}$$

In LIVARHACC (Figure 6), we have the following property to bound the size of a live variable set during local accumulation:

Lemma 2 *During local accumulation, the size of the local live variable set when processing φ_{i-j} is at most $j + 1$.*

Thus, in LIVARHACC, an upper bound on the total number of updates needed to preaccumulate the same SACs locally can be given by:

$$U_L = \sum_{j=0}^{k-1} \Big(j p_{i-j} + p_{i-j}(p_{i-j} + 1) \Big). \tag{14}$$

There is an important difference between Equation (14) and Equation (13). In Equation (14), we take into consideration that the case in Figure 5 a) and the case in Figure 5 b) are exclusive. In contrast, in Equation (13) (as well as in Equations (10) and (15)), the exclusiveness is ignored to simplify the equation.

Assume there are p "leaf" nodes in the local computational graph. Then an upper bound on the number of operations needed to globally accumulate the derivatives can be given by:

$$U_G = d_i p + p(p + 1). \tag{15}$$

Finally, under the assumption that all operators are unary or binary (which implies $p_{i-j} \leq 2, 0 \leq j < k$), the following two lemmas are easy to prove.

Lemma 3 *For an assignment and associated computational graph defined by $\varphi_{i-k+1}, \cdots, \varphi_i$, the relationship $\sum_{j=0}^{k-1} p_{i-j} \geq k + p - 1$ holds. Equality holds if and only if every variable on the left-hand-side is unique.*

Proof Assume there are q occurrences of explicitly declared variables in the assignment, where multiple occurrences of the same variable count independently. Obviously, $q \geq p$, and equality holds if and only if every variable on the left-hand-side is unique.

During the evaluation of this assignment, each SAC φ_{i-k+1} generates one variable (temporary result) and consumes p_{i-k+1} variables (operands). Initially, we have q variables on the right-hand-side as operands, and finally we have one variable on the left-hand-side as assignee. So we have $\sum_{j=0}^{k-1} p_{i-j} + 1 = q + k$. That is, $\sum_{j=0}^{k-1} p_{i-j} \geq k + p - 1$, and equality holds if and only if every variable on the left-hand-side is unique. □

Lemma 4 *For an assignment and associated computational graph defined by $\varphi_{i-k+1}, \cdots, \varphi_i$, the relationship $k \geq p$ holds. Equality holds if and only if every variable on the left-hand-side is unique, and every operator is binary except for the assignment operator "=".*

Proof Following the proof of Lemma 3, we know $\sum_{j=0}^{k-1} p_{i-j} + 1 = q + k$. Further, we have $p_i = 1$ because φ_i is an assignment operator. Further $p_{i-j} \leq 2, 1 \leq j < k$ because we assume all operators are unary or binary. Thus we have $p + k \leq q + k = \sum_{j=0}^{k-1} p_{i-j} + 1 \leq 2k$. That gives us $k \geq p$, and equality holds if and only if every variable on the left-hand-side is unique, and every operator is binary except for the assignment operator "=". □

Putting these results together, we prove Theorem 3 that we had stated in Section 4.2, and which we redisplay here (for convenience) as Theorem 4.

Theorem 4 *Given a statement defined by $\varphi_{i-k+1}, \cdots, \varphi_i$, when $d_i(k-1) \geq 2k(k+1)$ holds, we have $U_L + U_G \leq U$.*

Proof First we consider the left-hand-side of the inequality.

$$\begin{aligned}
U_L + U_G &= \sum_{j=0}^{k-1} (jp_{i-j} + p_{i-j}(p_{i-j}+1)) + d_i p + p(p+1) \\
&= d_i p + p(p+1) + \sum_{j=0}^{k-1} jp_{i-j} + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \\
&\leq d_i p + p(p+1) + 2\sum_{j=0}^{k-1} j + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \qquad (\text{since } p_{i-j} < 2) \\
&\leq d_i p + p(p+1) + k(k+1) + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \qquad (\text{upper bound for the sum on } k) \\
&\leq 2k(k+1) + d_i p + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \qquad (\text{By Lemma 4}, k \geq p).
\end{aligned}$$

Now we provide a lower bound for the right-hand-side of the inequality.

$$\begin{aligned}
U &= \sum_{j=0}^{k-1} \Big(d_{i-j} p_{i-j} + p_{i-j}(p_{i-j}+1) \Big) \\
&= \sum_{j=0}^{k-1} d_{i-j} p_{i-j} + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \\
&\geq d_i \sum_{j=0}^{k-1} p_{i-j} + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \qquad (\text{By Lemma 1}, d_i \leq d_{i-j}) \\
&\geq d_i(k+p-1) + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1) \qquad (\text{By Lemma 3}) \\
&= d_i(k-1) + d_i p + \sum_{j=0}^{k-1} p_{i-j}(p_{i-j}+1).
\end{aligned}$$

Thus when $d_i(k-1) \geq 2k(k+1)$, we have $U_L + U_G \leq U$. □

A.3 Listing of the Synthetic Test Functions `F1`, `F2`, `F3` and `F4`

We list in Table 15 the mathematical definitions of the four synthetic test functions `F1`, `F2`, `F3` and `F4` we used in the experiments discussed in Section 6. The functions are called in [20] Problem 1, Problem 5, Problem 80, and Problem 41, respectively.

Table 15 Mathematical descriptions of the synthetic test functions used in the experiments.

Chained Rosenbrock function (`F1`):

$$F(x) = \sum_{i=2}^{i=n} \left[100(x_{i-1}^2 - x_i)^2 + (x_{i-1} - 1)^2\right]$$

Generalized Broyden banded function (`F2`):

$$F(x) = \sum_{i=1}^{i=n} |(3 - 2x_i)x_i + \sum_{j \in J_i} x_j(1 + x_j)|^{\frac{7}{3}}$$
$$J_i = \{j | \min(1, i-5) \leq j \leq \max(n, i+1)\}$$

Potra-Rheinboldt boundary value problem (`F3`):

$$F(x) = \frac{1}{2} \sum_{k=1}^{k=n} f_k^2(x)$$
$$f_k(x) = 2x_k - x_{k-1} - x_{k+1} + h^2(x_k^2 + x_k + 0.1x_{k+n/2} - 1.2) \qquad 1 \leq k \leq n/2$$
$$f_k(x) = 2x_k - x_{k-1} - x_{k+1} + h^2(0.2x_{k-n/2} + x_k^2 + x_k - 0.6) \qquad n/2 < k \leq n, h = 1/(n/2+1)$$

Gomez-Ruggiero function (`F4`):

$$F(x) = \frac{1}{2} \sum_{k=1}^{k=n} f_k^2(x)$$
$$f_k(x) = -2x_k^2 + 3x_k + 2x_{k+1} + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \qquad k = 1$$
$$f_k(x) = -2x_k^2 + 3x_k - x_{k-1} + 2x_{k+1} + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \qquad 1 < k < n$$
$$f_k(x) = -2x_k^2 + 3x_k - x_{k-1} + 3x_{n-4} - x_{n-3} - x_{n-2} + 0.5x_{n-1} - x_n + 1 \qquad k = n$$

References

1. A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Number 105. SIAM, Philadelphia, PA, 2nd edition, 2008.
2. U. Naumann. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation.* SIAM, 2012.
3. R.H.F. Jackson and G.P. McCormick. *The polyadic structure of factorable function tensors with applications to higher-order minimization techniques.* J. Optim. Theory Appl. 51, pp 63–94, 1986.
4. B. Christianson. *Automatic Hessians by reverse accumulation.* IMA J. Numer. Anal 12(2), pp 135–150, 1992.
5. L.C.W. Dixon. *Use of Automatic Differentiation for calculating Hessians and Newton steps.* In *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, A. Griewank and G.F. Corliss, eds., SIAM, Philadelphia, 1991, pp 114–125.
6. S. Bhowmick and P.D. Hovland. *A polynomial-time algorithm for detecting directed axial symmetry in Hessian computational graphs.* In *Advances in Automatic Differentiation*, C. H. Bischof, H. M. Bucker, P.D. Hovland, U. Naumann and J. Utke, eds., Springer, Berlin, 2008, pp 91–102.
7. T. Coleman and J. Moré. *Estimation of sparse Hessian matrices and graph coloring problems.* Math. Program. 28, pp 243-270, 1984.
8. T. Coleman and J. Cai. *The cyclic coloring problem and estimation of sparse Hessian matrices.* SIAM J. Alg. Disc. Meth. 7(2), pp 221-235, 1986.
9. A. H. Gebremedhin, A. Tarafdar, F. Manne, and A. Pothen. *New acyclic and star coloring algorithms with applications to Hessian computation.* SIAM Journal on Scientific Computing, 29(3):1042–1072, 2007.
10. A. H. Gebremedhin, A. Tarafdar, A. Pothen, and A. Walther. *Efficient computation of sparse Hessians using coloring and Automatic Differentiation.* INFORMS J. on Computing, 1(2):209–223, 2009.

11. R. M. Gower and M. P. Mello. *A new framework for Hessian Automatic Differentiation.* Optimization Methods and Software, Volume 27, Issue 2, pages 233–249, 2012.
12. R. M. Gower and M. P, Mello. *Computing the Sparsity Pattern of Hessians using Automatic Differentiation.* ACM Transactions on Mathematical Software, Volume 40 Issue 2, Article No. 10, 2014.
13. R. M. Gower and A. L. Gower. *Higher-order Reverse Automatic Differentiation with Emphasis on the Third-Order.* Mathematical Programming, ISSN: 0025-5610, pages 1–23, 2014.
14. A. Giewank, D. Juedes and J. Utke. *ADOL-C: A package for the Automatic Differentiation of algorithms written in C/C++.* ACM Trans. Math. Softw. 22, 131-167, 1996.
15. A. Walther and A. Griewank. *Getting started with ADOL-C.* In *Combinatorial Scientific Computing*, Chapman-Hall CRC Computational Science. Chapter 7, 181-202, 2012.
16. L. Hascoët, U. Naumann and V. Pascual. *"To Be Recorded" Analysis in Reverse-Mode Automatic Differentiation.* Future Generation Comput. Syst. 21: 1401–1417, 2005.
17. L. Hascoët and M. Araya-Polo. *The Adjoint Data-Flow Analyses: Formalization, Properties, and Applications.* In M. Bücker, G.Corliss, P. Hovland, U. Naumann and B. Norris, editors, *Automatic Differentiation: Applications, Theory and Tools*, Lecture Notes in Comput. Sci. Engr. 50, Springer, Berlin, 2005, pages 135–146, 2005.
18. A. Griewank. *Sequential Evaluation of Adjoints and Higher Derivative Vectors by Overloading and Reverse Accumulation.* Research Report Konrad-Zuse-Zentrum for Informationstechnik Berlin, No 0, 1991.
19. J. Utke *Flattening of Basic Blocks for Preaccumulation* Automatic Differentiation: Applications, Theory, and Implementations, Springer, 2006, pages 121–133, 2006.
20. L. Luksan, C. Matonoha and J. Vlcek: *Sparse Test Problems for Unconstrained Optimization.* Tech. Rep. V-1064, ICS AS CR, January 2010.
21. T. S. Munson and P. D. Hovland. *The FeasNewt benchmark.* Workload Characterization Symposium, 2005. Proceedings of the IEEE International, pages 150-154, 2005
22. A. H. Gebremedhin, D. Nguyen, M. Patwary, and A. Pothen. *ColPack: software for graph coloring and related problems in scientific computing.* ACM Trans. Math. Sofw. 40(1), pp 1–31, 2013.
23. A. Walther. *Computing sparse Hessians with Automatic Differentiation.* ACM Trans. Math. Softw. 34(1), pp 1–15, 2008.
24. A. Walther. *On the efficient computation of sparse patterns for Hessians.* S. Forth et al. (eds), Recent Advances in Algorithmic Differentiation, Lecture Notes in Computational Science and Engineering 87, pages 139–149, 2012.
25. N. Nicholas and S. Julian. *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation.* Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA, June 2007.