Computing Sparse Hessians with Automatic Differentiation

ANDREA WALTHER
Institute of Scientific Computing, Technische Universität Dresden

3

A new approach for computing a sparsity pattern for a Hessian is presented: nonlinearity information is propagated through the function evaluation yielding the nonzero structure. A complexity analysis of the proposed algorithm is given. Once the sparsity pattern is available, coloring algorithms can be applied to compute a seed matrix. To evaluate the product of the Hessian and the seed matrix, a vector version for evaluating second order adjoints is analysed. New drivers of ADOL-C are provided implementing the presented algorithms. Runtime analyses are given for some problems of the CUTE collection.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms: Algorithms, Theory, Performance

Additional Key Words and Phrases: Automatic differentiation, second order derivatives, sparsity pattern

ACM Reference Format:

Walther, A. 2008. Computing sparse hessians with automatic differentiation. ACM Trans. Math. Softw., 34, 1, Article 3 (January 2008), 15 pages. DOI = 10.1145/1322436.1322439 http://doi.acm.org/ 10.1145/1322436.1322439

1. INTRODUCTION

Several solvers for nonlinearly constrained optimization problems allow or even require the provision of exact second order derivatives [Vanderbei and Shanno 1999; Wächter and Biegler 2006; Waltz and Nocedal 2003]. Furthermore, exact second order derivatives are needed to compute parametric sensitivities, for example, for the real-time control of dynamical systems, see Büskens and Maurer [2001]. Quite often, the corresponding Hessians are sparse, for example due to the discretization of a differential equation describing the considered problem. To maintain the efficiency of the algorithms, it is important to take this sparsity information into account. Therefore, some of the tools [Vanderbei and Shanno

Authors' address: email: Andrea.Walther@tu.dresden.de

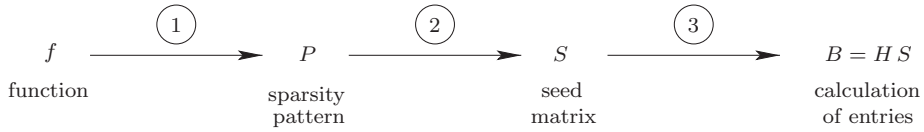

Fig. 1. Computing sparse Hessians.

1999; Wächter and Biegler 2006] assume that the user provides the Hessian in a sparse format.

As soon as a sparsity pattern for the Hessian is known, well-established coloring algorithms [Coleman and Moré 1984; Gebremedhin et al. 2005], in combination with Automatic Differentiation (AD) [Griewank 2000] allow the efficient computation of the required second-order information. The overall process is illustrated in Figure 1. Ideally, steps 1 and 2 of the process, that is, the generation of the sparsity pattern P and the calculation of the so-called seed matrix S using graph coloring have to be performed only once. Subsequently, the entries of the sparse Hessian can be computed in a compressed form using the second order adjoint mode of AD. The knowledge of the sparsity pattern P is essential for the approach sketched in Figure 1.

So far, only AMPL [Gay 1996] can compute structural information about the Hessian automatically. For that purpose, the partial separability of the differentiated function is exploited. In this paper, we propose and analyse a new algorithm for computing a sparsity pattern P. As an alternative to the present work, the calculation of sparse Hessians may also rely on elimination rules for the computational graph of the Hessian. This approach was first considered in Dixon [1991] and is the subject of current research. Similar techniques are well-established for the computation of the complete Jacobian [Naumann 2002; Naumann 2004].

We assume throughout that the function $f : \mathbb{R}^n \to \mathbb{R}, \quad x \mapsto y$, to be differentiated is at least twice continuously differentiable and given as a computer program in an imperative programming language. Then, the Hessian of f at a given point x defined by

$$H(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1 \partial x_1}(x) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(x) \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(x) & \cdots & \frac{\partial^2 f}{\partial x_n \partial x_n}(x) \end{pmatrix}$$

is a symmetric matrix. Due to the second order derivatives, an entry

$$H_{ij}(x) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x)$$

in the Hessian can only be nonzero if the computation of $y = f(x)$ involves a term that depends nonlinearly on both x_i and x_j. In this paper, we propose a new algorithm that propagates appropriate nonlinearity information through the function computation. Subsequently, a sparsity pattern for the Hessian can be derived directly from the propagated index sets. For any AD-tool based on

Table I. Formalization of Evaluation

Algorithm I: Function evaluation
for $i = 1, \ldots, n$
$\quad v_{i-n} = x_i$
for $i = 1, \ldots, l$
$\quad v_i = \varphi_i(v_j)_{j \prec i}$
$y = v_l$

operator overloading, one can implement the proposed approach easily just as a new variant of the derivative calculation. In this article, we present a new driver function of the AD-tool ADOL-C [Walther et al. 2005] to compute the required sparsity pattern. Then, we generate a seed matrix S to compute the entries of the sparse Hessian by applying a graph coloring algorithm first proposed in Coleman and Moré [1984]. Subsequently, we present a vector mode for computing second-order adjoint information. This vector version avoids the recomputation of intermediate results and reduces the cost to evaluate the Hessian-matrix product $H(x)S$ significantly. The proposed approach is implemented as a recent driver of ADOL-C that allows for the first time the computation of Hessian-matrix products instead of only Hessian-vector products.

This article has the following structure. In Section 2, we introduce the function representation that is used to derive and analyze the proposed computation of a sparsity pattern. Subsequently, the propagation of nonlinearity is presented, and a complexity analysis for the new algorithm is given. Section 3 sketches very briefly the graph coloring approach for generating the seed matrix S. Furthermore, it describes a new driver of ADOL-C implementing this algorithm. In Section 4, we present and analyse a vector version of the second order adjoint mode of AD. The corresponding implementation in ADOL-C is sketched. This includes also a new algorithm to compute the Hessian in a sparse format. Section 5 contains runtime analyses to verify the complexity results. Finally, we draw some conclusions and give an outlook in Section 6.

2. COMPUTING A SPARSITY PATTERN

2.1 Function Representation

Throughout, we assume that the calculation of $y = f(x)$ can be split into a presumably very long sequence of unary or binary operations. A formalization of the function evaluation similar to the one introduced in Griewank [2000] is shown in Table I. The first loop copies the current values of the independent variables $x_1, \ldots, x_n$ into the internal variables $v_{1-n}, \ldots, v_0$. The function evaluation itself consisting of l unary or binary operations is performed in the second loop. Finally the value of the dependent variable y is extracted from the corresponding internal variable v_l. As can be seen, each intermediate value v_i with $1 \leq i \leq l$ is computed by applying an *elemental function* φ_i. The function φ_i may have one or two arguments identified by the *precedence relation* $j \prec i$, where we have $\varphi_i(v_j)_{j \prec i} = \varphi_i(v_j)$ or $\varphi_i(v_j)_{j \prec i} = \varphi_i(v_j, v_{\hat{j}})$ with $j < i$ and $j, \hat{j} < i$, respectively. Hence, the precedence relation $j \prec i$ denotes that v_i depends directly on v_j.

Since we assume that f is at least twice continuously differentiable, the set of elemental functions may comprise simple evaluations, for example, additions, multiplications, and calls to intrinsic functions such as $\sin(x)$ or $\exp(x)$ provided by a high-level computer language like Fortran or C such that they are two times differentiable. The approach presented below can be extended to piecewise-differentiable functions like $\max(v_j, v_j)$ or $\sqrt{v_j}$ as long as these elemental functions are evaluated on the differentiable parts.

2.2 Propagation of Nonlinear Interaction

Based on the decomposition into elemental functions, one can now define two different index sets to propagate nonlinearity information through the function evaluation. First, we will need *index domains*

$$\mathcal{X}_k \equiv \{j \leq n \,:\, j - n \prec^* k\} \quad \text{for} \quad 1 - n \leq k \leq l$$

for all intermediate variables v_k as already defined in Griewank [2000, Section 6.1]. Here, $\prec^*$ denotes the transitive closure of the precedence relation $\prec$. One can compute the index domains using the forward recurrence

$$\mathcal{X}_k = \bigcup_{j \prec k} \mathcal{X}_j \quad \text{from} \quad \mathcal{X}_{j-n} = \{j\} \quad \text{for} \quad 1 \leq j \leq n.$$

This approach yields the inclusion

$$\left\{ j \leq n \,:\, \frac{\partial v_k}{\partial x_j} \not\equiv 0 \right\} \subseteq \mathcal{X}_k$$

and identity will hold as long as no degeneracy occurs. One example for a proper subset relation is given by the statement sequence

$$v_1 = \sin(v_0), \quad v_2 = \cos(v_0), \quad v_3 = v_1 * v_1, \quad v_4 = v_2 * v_2, \quad v_5 = v_3 + v_4$$

as mentioned already in [Griewank 2000]. Obviously, one has $\partial v_5/\partial v_0 = \partial v_5/\partial x_1 = 0$ but $\mathcal{X}_5 = \mathcal{X}_4 = \mathcal{X}_3 = \mathcal{X}_2 = \mathcal{X}_1 = \{1\}$. For the complexity analysis given in Sec. 2.3, we define $\bar{n}_k \equiv |\mathcal{X}_k|$ for all $1 - n \leq k \leq l$.

The index domains $\mathcal{X}_k$ belonging to the dependent variables can be used to exploit sparsity for the computation of Jacobian matrices as explained in Griewank [2000, Chapter 7]. However, we want to go one step further in computing a sparsity pattern for the Hessian. Therefore, we need additional index sets $\mathcal{N}_i$, $1 \leq i \leq n$, called *nonlinear interaction domains* (NID) for all independent variables, such that

$$\left\{ j \leq n \,:\, \frac{\partial^2 y}{\partial x_i \partial x_j} \not\equiv 0 \right\} \subseteq \mathcal{N}_i \,. \tag{1}$$

Once more, degeneracies may cause a proper subset relation in (1). In the case of second-order derivatives considered here, degeneracy may, for example, arise through statement sequences such as $y = x(\sin^2(x) + \cos^2(x))$ given by

$$v_1 = \sin(v_0), \quad v_2 = \cos(v_0), \quad v_3 = v_1 * v_1, \quad v_4 = v_3 * v_0, \quad v_5 = v_2 * v_2,$$
$$v_6 = v_5 * v_0, \quad v_7 = v_6 + v_4.$$

Table II. Propagation of Nonlinear Interaction

Algorithm II: Computation of nonlinear interaction domains

for $i = 1, \ldots, n$
  $\mathcal{X}_{i-n} \leftarrow \{i\}, \quad \mathcal{N}_i \leftarrow \emptyset$
for $i = 1, \ldots, l$
  $\mathcal{X}_i \leftarrow \bigcup_{j \prec i} \mathcal{X}_j$ (2)
  if φ_i **nonlinear then**
    if $v_i = \varphi_i(v_j)$ **then**
      $\forall k \in \mathcal{X}_i : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$ (3)
    if $v_i = \varphi_i(v_j, v_j)$ **then**
      if v_i **linear in** v_j **then**
        $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j$ (4)
      else
        $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$ (5)
      if v_i **linear in** v_j **then**
        $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j$ (6)
      else
        $\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i$ (7)

Then, one has $\partial^2 v_7 / \partial v_0^2 = \partial^2 v_7 / \partial x_1^2 = 0$ but $\mathcal{N}_1 = \{1\}$.

After setting $\mathcal{N}_i = \emptyset$ at the beginning of the function evaluation, the NIDs have to be updated for each nonlinear operation that occurs during the function evaluation such that $\mathcal{N}_i$ contains the indices $1 \leq j \leq n$ of all independents that are combined in a nonlinear fashion with the independent x_i. This results in the algorithm shown by Table II. As can be seen from Algorithm II, dead ends, that is, intermediate variables v_k that were computed but not needed subsequently to calculate v_l, can be contained in the function evaluation. These dead ends may cause that identity does not hold in (1).

Such dead ends have no consequences for the index domains $\mathcal{X}_k$, since they are defined for each intermediate value v_k. Hence, if a dead end is contained in the code, the corresponding index domains would be computed but they would have no influence on the index domain of the dependent variable y. This does not hold for the nonlinear interaction domains $\mathcal{N}_i$, since the $\mathcal{N}_i$ are defined only for the independent variables x_i. Hence, if a dead end occurs the $\mathcal{N}_i$ would be extended despite the fact that the computed values have no influence on the dependent variable y. This would result in an overestimate of the sparsity pattern.

To illustrate the algorithm, we will consider the function

$$f : \mathbb{R}^6 \to \mathbb{R}, \quad f(x) = \sin(x_1 x_2) + \cos(x_3 + x_4) + 3(x_5 + x_6).$$

Table III shows the function evaluation and the development of the index sets $\mathcal{X}_i$ and $\mathcal{N}_i$ applying Algorithm II. As can be seen, the nonzero entries of the row i or column i of the Hessian $H(x)$ are given by the indices contained in the NIDs $\mathcal{N}_i$ for all $1 \leq i \leq 6$.

Table III. Function Evaluation and Execution of Algorithm II

for $i = 1, \ldots, 6$			
$\mathcal{X}_{i-n} = \{i\}, \quad \mathcal{N}_i = \emptyset$			
$v_1 = v_{-5} * v_{-4},$	$\mathcal{X}_1 = \{1, 2\},$	$\mathcal{N}_1 = \{2\},$	$\mathcal{N}_2 = \{1\}$
$v_2 = \sin(v_1),$	$\mathcal{X}_2 = \{1, 2\},$	$\mathcal{N}_1 = \{1, 2\},$	$\mathcal{N}_2 = \{1, 2\}$
$v_3 = v_{-3} + v_{-2},$	$\mathcal{X}_3 = \{3, 4\}$		
$v_4 = \cos(v_3),$	$\mathcal{X}_4 = \{3, 4\},$	$\mathcal{N}_3 = \{3, 4\},$	$\mathcal{N}_4 = \{3, 4\}$
$v_5 = v_{-1} + v_0,$	$\mathcal{X}_5 = \{5, 6\}$		
$v_6 = 3 * v_5,$	$\mathcal{X}_6 = \{5, 6\}$		
$v_7 = v_2 + v_4,$	$\mathcal{X}_7 = \{1, 2, 3, 4\}$		
$v_8 = v_7 + v_6,$	$\mathcal{X}_8 = \{1, 2, 3, 4, 5, 6\}$		

Table IV. Algorithm Used to Merge Two Sets

Algorithm III: Merging of two sets

1. For each i in the first set, put i in the new set and set flag(i) to false.
2. For each i in the second set, put i in the new set if flag(i) is true.
3. For each i in the first set, set flag(i) to true.

2.3 Complexity Analysis

First, one has to note that an implementation of Algorithm II based on operator overloading does not require the coding of the if-statements. Here, the corresponding set operations can be coded together with or instead of the elemental function evaluation. Therefore, we ignore the if-statements in the following complexity analysis. To ensure that the proposed algorithm provides an efficient method to compute a sparsity pattern for the Hessian $H(x)$, we have to examine the set operations (2)–(7) of Algorithm II. These merging operations are performed as described in Table IV, where flag denotes a logical array of length n that is set to true.

For the complexity result proved in the following theorem, we define the execution of one of the loop bodies as one operation MERGE. Hence, the operation count of the merging procedure is twice the length of the first list plus the length of the second list. It is possible to prove the following result:

THEOREM 2.1 (COMPLEXITY RESULT FOR ALGORITHM II). *Let OPS(NID) denote the number of operations MERGE needed by Algorithm II to generate all* $\mathcal{N}_i$, $1 \le i \le n$. *Then, the inequality*

$$OPS(NID) \le 6(1 + \hat{n}) \sum_{i=1}^{l} \bar{n}_i \tag{10}$$

is valid, where l *is the number of elemental functions evaluated to compute the function value of* f, $\bar{n}_i = |\mathcal{X}_i|$, *and* $\hat{n} = \max_{1 \le i \le n} |\mathcal{N}_i|$.

PROOF. The set operation (2) is either $\mathcal{X}_i \leftarrow \mathcal{X}_j$ if φ_i is unary or $\mathcal{X}_i \leftarrow \mathcal{X}_j \cup \mathcal{X}_{\hat{j}}$ if φ_i is binary. Therefore, we obtain for the operation count of the set operation (2) that

$$\mathrm{OPS}\Big(\mathcal{X}_i \leftarrow \bigcup_{j \prec i} \mathcal{X}_j\Big) \le 3\bar{n}_i \,. \tag{11}$$

Furthermore, the number of elements in each NID $\mathcal{N}_i$, $1 \leq i \leq n$, can be bounded by

$$|\mathcal{N}_i| \leq \hat{n} \tag{12}$$

due to the definition of $\hat{n}$. Furthermore, we can conclude for the set operations (3), (5), and (7) that

$$|\mathcal{X}_i| = \bar{n}_i \leq \hat{n} \tag{13}$$

if v_i is the result of a nonlinear operation because of (12). Due to the same reason, we obtain for the set operations (4) and (6)

$$|\mathcal{X}_j| = \bar{n}_j \leq \hat{n} \quad \text{and} \quad |\mathcal{X}_j| = \bar{n}_j \leq \hat{n}\ , \tag{14}$$

respectively. Using (13) and (14), it follows for the set operations (3)–(7) that

$$\begin{aligned}
\text{OPS}(\forall k \in \mathcal{X}_i : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i) \leq &\quad \bar{n}_i(2\hat{n} + \bar{n}_i) \leq \quad \bar{n}_i(2\hat{n} + \hat{n}) = \quad 3\hat{n}\bar{n}_i \\
\text{OPS}(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j) \leq &\quad \bar{n}_j(2\hat{n} + \bar{n}_j) \leq \quad \bar{n}_i(2\hat{n} + \hat{n}) = \quad 3\hat{n}\bar{n}_i \\
\text{OPS}(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i) \leq &\quad \bar{n}_j(2\hat{n} + \bar{n}_i) \leq \quad \bar{n}_i(2\hat{n} + \hat{n}) = \quad 3\hat{n}\bar{n}_i \\
\text{OPS}(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_j) \leq &\quad \bar{n}_j(2\hat{n} + \bar{n}_j) \leq \quad \bar{n}_i(2\hat{n} + \hat{n}) = \quad 3\hat{n}\bar{n}_i \\
\text{OPS}(\forall k \in \mathcal{X}_j : \mathcal{N}_k \leftarrow \mathcal{N}_k \cup \mathcal{X}_i) \leq &\quad \bar{n}_j(2\hat{n} + \bar{n}_i) \leq \quad \bar{n}_i(2\hat{n} + \hat{n}) = \quad 3\hat{n}\bar{n}_i\ .
\end{aligned}$$

The set operation (2) has to be executed exactly once for the calculation of the intermediate v_i, $1 \leq i \leq l$. Furthermore, at most two of the set operations (3)–(7) have to be executed. This yields the overall bound (10) due to the assumption of unary and binary operations. □

2.4 Computing Sparsity Patterns

The AD-tool ADOL-C was augmented with the new driver

```
int hess_pat(short tag, int n, double* x, unsigned int** P, int option);
```

to compute a sparsity pattern for the Hessian for a given function according to Algorithm II. The first argument, that is, tag, identifies the internal representation for which one wants to compute derivative information [Walther et al. 2005]. The next argument is used for a consistency check comparing this value to the one that is stored in the internal representation. The third argument, that is, x, defines the point for which a sparsity pattern for the Hessian is computed. After the function call P contains a sparsity pattern for the Hessian, where P[j][0] contains the number of nonzero elements in the jth row. The components P[j][i], $0 <$ i $\leq$ P[j][0], store the indices of these entries. The usage of the routine hess_pat is described in more detail in Walther et al. [2005] including information about allocation and deallocation schemes.

Obviously, the sparsity pattern P may vary as a function of the independent variable vector x for one of the following three reasons:

(A) Numerical values may be incidentally zero,
(B) fmin, fmax or other conditional assignments may flip to a different branch and
(C) the control flow may be completely changed.

According to the algorithm presented in this section, ADOL-C propagates generic dependencies and disregards incidental zeros that are due to cancellations or special values of the independent variables (case (A)). The treatment of case (B) is determined by the last argument option of the new driver. The default value is option = 0 resulting in a more conservative computation of a sparsity pattern for the Hessian. It accounts for all dependencies that might occur for any value of the independent variables. For example, the intermediate $v_i = \max(v_j, v_j)$ is always assumed to depend on all independent variables that v_j or v_j depend on and the index domain $\mathcal{X}_i$ is extended correspondingly. In contrast, the tight version option = 1 gives this result only in the unlikely event of an exact tie $v_j = v_j$. Otherwise it sets the index domain $\mathcal{X}_i$ either to $\mathcal{X}_j$ or to $\mathcal{X}_j$, depending on whether $v_i = v_j$ or $v_i = v_j$ locally. Obviously, a sparsity pattern obtained with the tight option may contain more zeros than the one obtained with the safe option. On the other hand, it will be valid only at points belonging to an area where the function f is locally smooth and that contains the point at which the internal representation was generated. Case (C) results in a negative return value of the new driver indicating that the internal representation of the given function is not valid for the current argument x due to a change in the control flow. Then, before computing a sparsity pattern one has to generate a new internal representation by retaping the function evaluation at x. Details can be found in the ADOL-C documentation [Walther et al. 2005].

3. COMPUTING THE SEED MATRIX

For computing sparse Jacobians, the application of compression techniques is now well-established. A comprehensive introduction to this approach can be found, for example, in Griewank [2000]. Naturally, the same idea can also be exploited for computing sparse Hessians. Hence, the entries of the sparse Hessian are computed by evaluating the product

$$B = H(x)S \in \mathbb{R}^{n\times q}$$

for a so-called seed matrix $S \in \mathbb{R}^{n\times q}$. Here, as simplest option the columns of S are chosen as vectors the entries of which are either 0 or 1. After the computation of the matrix B, one has to reconstruct the entries of $H(x)$ from the available derivative information; see, for example [Griewank 2000]. Depending on the choice of the seed matrix, this may require solving a linear system whose matrix is either a permutation of the identity or a triangular matrix. In the first case, the entries of $H(x)$ can be directly extracted from B. The resulting evaluation scheme is therefore called *direct*. In the second case one has to solve simple equations. Hence, the resulting evaluation scheme is called *substitution-based*.

A first method to generate a seed matrix S was proposed by Powell and Toint [1979]. Later, Coleman and Moré [1984] observed that the task of finding a suitable S is equivalent to a graph coloring problem, where the symmetry of the

derivative matrix can be exploited to reduce the required number q of columns in the seed matrix. The method proposed in Coleman and Moré [1984] can be seen as a relaxed distance-2 and a restricted distance-1 coloring. Therefore, it is referred to as distance-$\frac{3}{2}$ coloring in Gebremedhin et al. [2005]. This coloring method, recently studied also by Albertson et al. [2004], is now used by ADOL-C to generate the seed matrix S, yielding a direct evaluation scheme. As alternative one may consider an acyclic coloring as proposed in Coleman and Cai [1986] that gives a substitution-based evaluation scheme. This approach will be integrated into ADOL-C in the near future.

The new driver

```
int generate_seed_hess(int n, unsigned int** P, double*** S, int* q);
```

of ADOL-C has as input variables the number of independent variables n and a sparsity pattern P computed for example by the algorithm described in the last section or provided by the user. First, it performs a coloring of the adjacency graph defined by the sparsity pattern P. The number of colors needed for the coloring determines the number of columns q in the seed matrix. Subsequently, the function allocates the memory needed by S and initializes S according to the graph coloring. Additional information about the usage of generate_seed_hess including details about the specific memory management can be found in Walther et al. [2005].

4. EVALUATING HESSIAN-MATRIX PRODUCTS

For a scalar valued function $y = f(x)$ exact Hessian-vector products can be computed by differentiating formally the results of the reverse mode of AD once more with respect to x and $\bar{y}$ using the scalar forward mode of AD. Using the notation introduced in Griewank [2000], this evaluation of second order adjoints is given by

$$y = f(x) \xrightarrow[\text{reverse}]{} \bar{x} = \bar{y}\, f'(x) \xrightarrow[\text{forward}]{} \dot{\bar{x}} = \bar{y}\, f''(x)\dot{x} + \dot{\bar{y}}\, f'(x) \in \mathbb{R}^n, \tag{15}$$

for $\bar{x}, \dot{x}, \dot{\bar{x}} \in \mathbb{R}^n$ and $\bar{y}, \dot{\bar{y}} \in \mathbb{R}$. Hence, the desired Hessian-vector product $f''(x)\dot{x}$ can be computed by setting $\bar{y} = 1$ and $\dot{\bar{y}} = 0$. As shown in Griewank [2000], Section 4.5, one obtains the following complexity estimate for the evaluation of a Hessian-vector product

$$\text{TIME}(f''(x)\dot{x}) \leq \omega_{soad}\text{TIME}(f(x)) \quad \text{with} \quad \omega_{soad} \in [7,\ 10].$$

Evaluating the second order adjoint procedure denoted by the subscript $soad$ for the p columns which form the seed matrix, we obtain as complexity estimate for evaluating $H(x)S$

$$\text{TIME}(H(x)S) \leq \omega_{soad}\, p\, \text{TIME}(f(x)) \quad \text{with} \quad \omega_{soad} \in [7,\ 10]. \tag{16}$$

Again setting $\dot{\bar{y}} = 0$, that is, ignoring the second term for computing $\dot{\bar{x}}$ in (15), and applying the vector forward mode for a given matrix $\dot{X} \in \mathbb{R}^{n\times p}$, one can similarly derive a vector version of the second order adjoint computation given by

$$y = f(x) \xrightarrow[\text{reverse}]{} \bar{x} = \bar{y}\, f'(x) \xrightarrow[\text{forward}]{} \dot{\bar{X}} = \bar{y}\, f''(x)\dot{X} \in \mathbb{R}^{n\times p}$$

Table V. Second-Order Adjoint Vector Evaluation

Algorithm IV: Evaluation of Hessian-matrix product

for $i = 1, \ldots, n$

$v_{i-n} = x_i, \quad \dot{V}_{i-n} = \dot{X}_i, \bar{v}_{i-n} = 0, \quad \dot{\bar{V}}_{i-n} = 0$

for $i = 1, \ldots, l$

$v_i = \varphi_i(v_j)_{j \prec i}, \quad \dot{V}_i = \sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \dot{V}_j, \quad \bar{v}_i = 0, \quad \dot{\bar{V}}_i = 0$

$y = v_l, \quad \dot{Y} = \dot{V}_l, \quad \bar{v}_l = \bar{y}$

for $i = l, \ldots, 1$

$\bar{v}_j \mathrel{+}= \bar{v}_i \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \quad \text{for} \quad j \prec i$

$\dot{\bar{V}}_j \mathrel{+}= \bar{v}_i \sum_{k \prec i} \frac{\partial^2}{\partial v_j \partial v_k} \varphi_i(v_j)_{j \prec i} \dot{V}_k + \dot{\bar{V}}_i \frac{\partial}{\partial v_j} \varphi_i(v_j)_{j \prec i} \quad \text{for} \quad j \prec i$

for $i = 1, \ldots, n$

$\bar{x}_i = \bar{v}_{i-n}, \dot{\bar{X}}_i = \dot{\bar{V}}_{i-n}$

Table VI. Second-Order Adjoint Vector Complexity

	Elemental function φ			
soadp	const	add/sub	mult	ψ
MOVES	2 + 2p	12 + 6p	11 + 11p	7 + 7p
ADDS	0	3 + 3p	2 + 5p	1 + 2p
MULTS	0	0	3 + 6p	1 + 4p
NLOPS	0	0	0	4

for $\bar{x} \in \mathbb{R}^n$, $\dot{X} \in \mathbb{R}^{n \times p}$, and $\bar{y} \in \mathbb{R}$. Omitting the storage of intermediate values for simplicity, the corresponding evaluation procedure is given in Table V.

To analyze the computation effort needed to evaluate $H(x)S$, we will use the complexity analysis introduced in Griewank [2000, Section 2.5]. Denoting the vector version of the second order adjoint computation with $soadp$, we obtain for assigning a constant, an addition or subtraction, a multiplication, and a general nonlinear function ψ as elemental function φ the operation counts given in Table VI, which can be directly derived from the complexity counts given in Griewank [2000, Tables 3.6, 4.11]. Here, MOVES denotes the number of memory accesses. From the stated operation counts, we can derive the runtime estimate

$$\text{TIME}(\dot{\bar{X}}) \leq \omega_{soadp} \text{TIME}(f(x)) \tag{17}$$

according to Griewank [2000, Section 2.5] with

$$\omega_{soadp} = \max \left\{ 2 + 2p, \frac{(12 + 6p)\mu + 3 + 3p}{3\mu + 1}, \frac{(11 + 11p)\mu + 2 + 5p + (3 + 6p)\pi}{3\mu + \pi}, \frac{(7 + 7p)\mu + 1 + 2p + (1 + 4p)\pi + 4\nu}{2\mu + \nu} \right\} \in [4 + 3p, 4 + 6p]. \tag{18}$$

The constants μ, π, and ν measure the complexity of a memory access, a multiplication, and a nonlinear operation, respectively, where the complexity

of an addition is normalized to 1. The reduction in the runtime ratio from an upper bound in $[7p, 10p]$ to an upper bound in $[4 + 3p, 4 + 6p]$ is caused by the fact that values that are independent of the directions contained in $\dot{X}$ are reused instead of recomputed. Hence, similar to the runtime reductions that can be achieved by using the vector forward mode of AD instead of the scalar forward mode for computing first derivatives, a decrease of the computing time needed for directional second derivatives can be achieved by using a vector version. The new ADOL-C driver

```
int hess_mat(short tag, int n, int p, double* x, double** S, double** B);
```

implements the vector version of the second-order adjoint computation. The inputs are the identifier for the internal representation tag, the number of independent variables n for a consistency check, the current value of the independent variables x, and the seed matrix S. The result of the product $H(x)S$ is stored as output of the function call in the two-dimensional array B of size $n \times p$. More information about hess_mat including details about the memory allocation can be found in Walther et al. [2005]. Using the three new ADOL-C drivers, it is possible to compute sparse Hessians in an efficient way as we will see in the next section. Since the Hessian entries are often required in a prescribed sparse format, ADOL-C also provides a new driver that computes the sparse Hessian and stores the entries directly in coordinate format:

```
int sparse_hess(short tag, int n, int repeat, double* x, int* nnz,
                unsigned int** r_ind, unsigned int** c_ind, double** H_val);
```

Once more, the input variables are the identifier for the internal representation tag, the number of independent variables n for a consistency check, the current value of the independent variables x. Furthermore, the flag repeat=0 indicates that a new seed matrix S has to be computed, whereas repeat=1 results in the reusage of the previously computed seed matrix. The input/output variable nnz stores the number of the nonzero entries. Therefore, nnz also denotes the length of the arrays r_ind storing the row indices, c_ind storing the column indices, and H_val storing the values of the nonzero entries. The manual [Walther et al. 2005] contains more information about the routine sparse_hess including details about the corresponding memory management.

5. NUMERICAL EXAMPLES

In this section, we will employ the AD-tool ADOL-C to present some runtime results for the proposed algorithms. For that purpose, we use optimization problems from the CUTE collection [Bongartz et al. 1995].

5.1 The Computation of Sparsity Patterns

In this subsection, we report on the run-time behaviour of the driver hess_pat described in Section 2.4 as an implementation of Algorithm II. As test cases, we chose the Lagrange function of the CUTE problems broydnbd ($\hat{n} = 1$), chainwoo ($\hat{n} = 4$), lminsurf ($\hat{n} = 9$), and morebv ($\hat{n} = 5$) with varying dimension n. For all four examples there exists at least one index domain that contains the indices of all independent variables, that is, there is at least one $i \in \{1, \ldots, l\}$

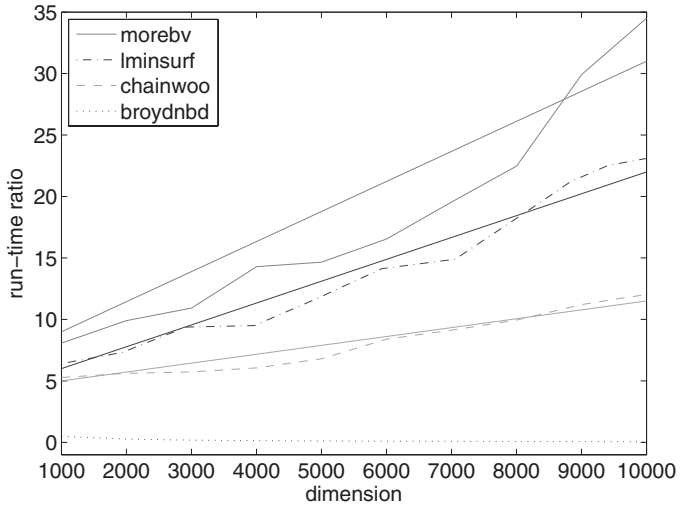

Fig. 2. Runtime results for Algorithm II.

with $\bar{n}_i = n$. This is possible because the bounds (13) and (14) hold only for intermediate variables that are the result and the argument of a nonlinear operation, respectively. Furthermore, almost all rows of the Hessians have $\hat{n}$ nonzero entries independent of the value of n. Throughout this subsection, the figures report the runtime ratio

$$\frac{\text{TIME}(\text{hess\_pat}(...))}{(1+\hat{n})\text{TIME}(f)}\,. \tag{19}$$

For n varying in the interval [1000, 10000] the runtime ratios obtained for the considered examples are illustrated in Figure 2. As can be seen, a constant runtime ratio is achieved for the broydnbd problem. For the problem chainwoo a small increase of the run-time ratio can be observed. For the problems lminsurf and morebv, we obtain a stronger increase of the runtime ratio (19). To analyze the linear behaviour in more detail, lines are add to the curves illustrating the runtime results. The slopes for the added lines are well below 0.003.

To analyse the runtime behavior of Algorithm II in more detail, we performed another test by varying the problem size and the number of nonzeros $\hat{n}$. For that purpose, we enlarged the original objective function by the additional term

$$\tilde{f}(x) = f(x) + \sum_{j=1}^{u} x_i x_{i+j}\,.$$

Hence, the number of nonzero diagonals in the Hessian of the Lagrange function can be varied by choosing u appropriately. We generated test cases for the problem chainwoo with $\hat{n} = 7, 9, 11$ instead of $\hat{n} = 4$ as in the original version. Furthermore, we studied test cases for the morebv problem with $\hat{n} = 7, 9, 11$ instead of $\hat{n} = 5$ as in the original version. Figure 3 illustrates the runtime ratios achieved. For both problems, the linear behavior of the runtime ratio is almost the same if the number of nonzeros is increased. To ease the interpretation of

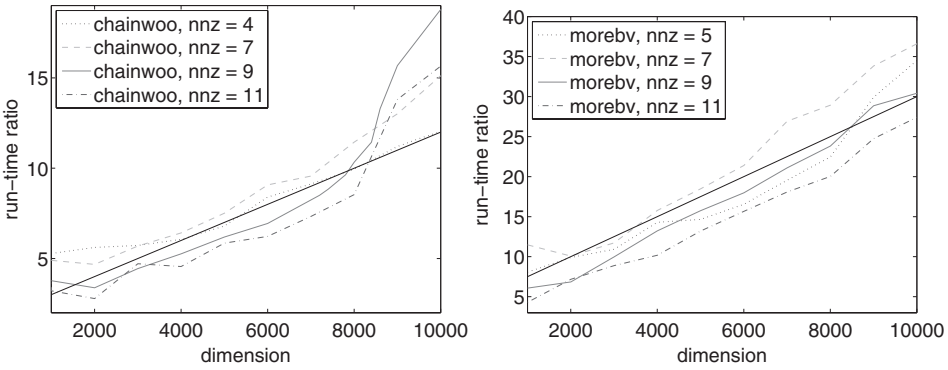

Fig. 3. Runtime results for Algorithm II and varying number of nonzeros.

the results, we added a black solid line with the slope 0.0001 for chainwoo and 0.0025 for morebv. As can be seen, the constant describing the linear increase of the runtime ratios is very small, which fits the expectations based on the complexity result presented in Section 2.3. Only for the problem chainwoo and $\hat{n} = 9$ and $\hat{n} = 11$ some cache effects disturbed the linear behavior in the range $n \in [8000, 9000]$. For the examples considered here, the linear increase with n of the complexity of Algorithm II can be described by a very small constant. Comparing the values observed for the runtime ratios with the complexity results given by (15) or (17), one finds that a sparsity pattern for the Hessian can be calculated at a cost that corresponds to the cost for computing a few columns of the Hessian itself.

5.2 The Evaluation of Hessian-Matrix Products

A second class of runtime tests was done for the newly proposed vector version of the second order adjoint mode. Comparing the complexity bounds for scalar second order adjoints given in (16) and for the vector version given in (17) and (18), one obtains

$$\frac{\text{TIME}(H(x)S)}{\text{TIME}(f(x))} \leq \omega_{soad}\, p \leq 10p \qquad \text{and} \qquad \frac{\text{TIME}(\dot{\tilde{X}})}{\text{TIME}(f(x))} \leq \omega_{soadp} \leq 4 + 6p,$$

respectively. We computed the stated runtime ratios using the well-established driver hess_vec of ADOL-C to compute p Hessian-vector products and the new driver hess_mat of ADOL-C to compute one Hessian-matrix product. The problems dtoc2 and eigena2 serve as test cases. For dtoc2 the corresponding seed matrix has 6 columns independent of the size of the problem. For the numerical tests we set the number of variables to $n = 1485$ and the number of constraints to $m = 990$. The number of columns that form the seed matrix varies with the problem size for eigena2. To get an impression also for a higher number of columns, we set the number of independent variables to $n = 2550$ and the number of constraints $m = 1275$, resulting in a seed matrix with 52 columns.

The achieved runtime ratios are illustrated by Figure 4. First of all, the expected linear behavior in dependence on the number of vectors and columns,

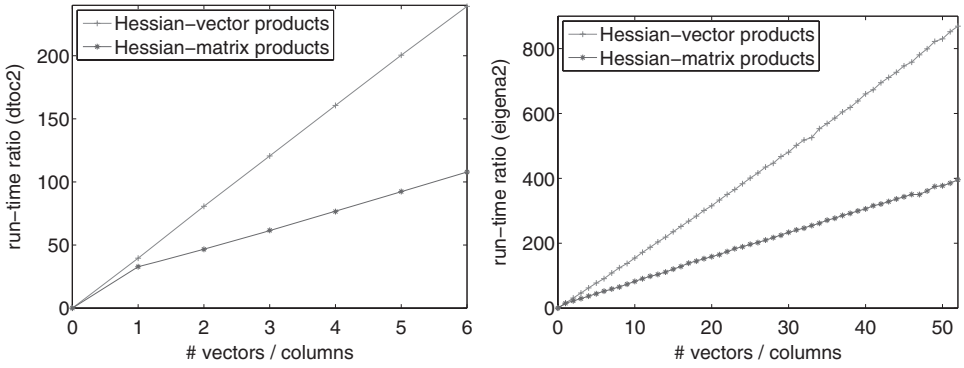

Fig. 4. Hessian-vector and Hessian-matrix products.

Table VII. Slopes Obtained from Runtime Results

	dtoc2	eigena2
scalar *soad*	28.6	16.5
vector *soad*	13.2	7.5

respectively, is clearly visible. Furthermore, the line with the larger slope belongs to the scalar second-order adjoint computation evaluating p Hessian-vector products. Hence, so far the theory is verified by the numerical examples, since the vector version of the second-order adjoint requires significantly less runtime.

Additionally, one can examine the slopes of the lines in more detail. For that purpose the slopes are stated in Table VII. As can be seen, the scalar mode is almost three times slower than the theory predicted for the dtoc2 example, whereas the vector mode is only about two times slower than the theory.

For the eigena2 example, the function evaluation is a little bit more complicated and the situation changes considerably in favor of ADOL-C. Here, the runtime needed by the scalar mode is only about a factor 3/2 larger than expected. So the operator-overloading tool ADOL-C comes almost close to the theory. The same is true for the vector version of the second order adjoint, where the slope is close to the theoretical bound 6.

6. CONCLUSION AND OUTLOOK

This article presents the propagation of nonlinearity for determining a sparsity pattern for a Hessian matrix. The complexity of the corresponding algorithm is analysed in detail. Once the sparsity pattern is available, well-known graph-coloring techniques can be applied to generate a seed matrix. Subsequently, the seed matrix can be used as input for a vector version of the second order adjoint mode, that is proposed and analysed in this paper for the first time. The three ingredients—sparsity pattern, seed matrix, vector second order adjoint computation—allow an efficient evaluation of sparse Hessians. Run-time results verifying the theoretical results are presented for some problems of the CUTE collection, where the AD-tool ADOL-C is used to compute the derivatives.

Future work can be devoted to the incorporation of more sophisticated coloring techniques which are the subject of current research. This includes also the incorporation of a substitution-based seeding.

ACKNOWLEDGMENTS

The author is grateful to Arijit Tarafdar and Assefaw Gebremedhin, Old Dominion University, for providing their graph coloring code. Furthermore, the author would like to thank the anonymous referees for their very valuable comments and constructive suggestions.

REFERENCES

ALBERTSON, A., CHAPPELL, G., KIERSTEAD, H., KÜNDGEN, A., AND RAMAMURTHU, R. 2004. Coloring with no 2-colored P_4's. *Electron. J. Comb. 11,* 1, R26.

BONGARTZ, I., CONN, A., GOULD, N., AND TOINT, P. 1995. CUTE: Constrained and unconstrained testing environment. *ACM Trans. Math. Softw. 21*, 123–160.

BÜSKENS, C. AND MAURER, H. 2001. Sensitivity analysis and real-time optimization of parametric nonlinear programming problems. In *Online Optimization of Large Scale Systems: State of the Art*, M. Gröschel, S. Krumke, and J. Rambau, Eds. Springer, 3–16.

COLEMAN, T. AND CAI, J. 1986. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth. 7*, 221–235.

COLEMAN, T. AND MORÉ, J. 1984. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program. 28*, 243–270.

DIXON, L. 1991. Use of automatic differentiation for calculating Hessians and Newton steps. In *Automatic Differentiation of Algorithms, Proceedings of the 1st SIAM Workshop on AD*. A. Griewank and G. Corliss, Eds. 114–125.

GAY, D. 1996. More AD of nonlinear AMPL models: Computing Hessian information and exploiting partial separability. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, Eds. SIAM, 173–184.

GEBREMEDHIN, A., MANNE, F., AND POTHEN, A. 2005. What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Revi. 47,* 4, 629–705.

GRIEWANK, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Frontiers in Applied Mathematics, 19. SIAM, Philadelphia, PA.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. 22*, 131–167.

NAUMANN, U. 2002. Cheaper Jacobians by simulated annealing. *SIAM J. Optim. 13*, 660–674.

NAUMANN, U. 2004. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Program. 99A*, 399–421.

POWELL, M. AND TOINT, P. 1979. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal. 16*, 1060–1074.

VANDERBEI, R. AND SHANNO, D. 1999. An interior-point algorithm for nonconvex nonlinear programming. *Comput. Optim. Appl. 13*, 231–252.

WÄCHTER, A. AND BIEGLER, L. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program. 106,* 1, 25–57.

WALTHER, A., KOWARZ, A., AND GRIEWANK, A. 2005. *Documentation of ADOL-C*: version 1.10.1. Updated version of Griewank et al. [1996].

WALTZ, R. AND NOCEDAL, J. 2003. KNITRO user's manual. Tech. rep. OTC 05/2003, Optimization Technology Center, Northwestern University, Evanston, IL.

Received April 2005; revised November 2005, October 2006, March 2007; accepted April 2007