# Exploiting Sparsity in Jacobian Computation via Coloring and Automatic Differentiation: A Case Study in a Simulated Moving Bed Process

Assefaw H. Gebremedhin[1] and Alex Pothen[1] and Andrea Walther[2]

[1] Department of Computer Science and Center for Computational Sciences, Old Dominion University, Norfolk, VA, USA, [assefaw,pothen]@cs.odu.edu
[2] Department of Mathematics, Technische Universität Dresden, Germany, andrea.walther@tu-dresden.de

**Summary.** Using a model from a chromatographic separation process in chemical engineering, we demonstrate that large, sparse Jacobians of fairly complex structures can be computed accurately and efficiently by using automatic differentiation (AD) in combination with a four-step procedure involving matrix *compression* and *de-compression*. For the detection of sparsity pattern (step 1), we employ a new operator overloading-based implementation of a technique that relies on propagation of *index domains*. To obtain the *seed* matrix to be used for compression (step 2), we use a *distance-2 coloring* of the bipartite graph representation of the Jacobian. The compressed Jacobian is computed using the vector forward mode of AD (step 3). A simple routine is used to directly recover the entries of the Jacobian from the compressed representation (step 4). Experimental results using ADOL-C show that the runtimes of each of these steps is in complete agreement with theoretical analysis, and the total runtime is found to be only about a *hundred* times the time needed for evaluating the function itself. The alternative approach of computing the Jacobian without exploiting sparsity is infeasible.

**Key words:** Sparse Jacobians, Graph coloring, Sparsity patterns, Simulated Moving Bed chromatography

## 1 Introduction

Automatic Differentiation (AD) has become a well established method for computing derivative matrices accurately and reliably. This work focuses on a set of techniques that constitute a scheme for making such a computation *efficient* in the case where the derivative matrix is large and sparse. The target scheme, outlined in Algorithm 1 in its general form, has been found to be an effective framework for computing Jacobian as well as Hessian matrices [2, 7]. The input to Algorithm 1 is a function $F$ whose derivative matrix $A \in \mathbb{R}^{m \times n}$ is sparse. The seed matrix $S$ determined in the second step of the algorithm is such that $s_{jk}$, its $(j,k)$ entry, is one if the $j$th column of the matrix $A$ belongs to group $k$ and zero otherwise. Since this corresponds to a partitioning of the columns of $A$, in every row $r$ of the matrix $S$ there is exactly one column $c$ in which the entry $s_{rc}$ is equal to one. There exist approaches that use a seed matrix where a row-sum is not necessarily equal to one [11], but they will not be considered here.

**Algorithm 1** A scheme for computing a sparse derivative matrix.

**procedure** SPARSECOMPUTE($F : \mathbb{R}^n \to \mathbb{R}^m$)

1. Determine the *sparsity structure* of the derivative matrix $A \in \mathbb{R}^{m \times n}$ of $F$.
2. Using a *coloring* on an appropriate graph of $A$, obtain an $n \times p$ *seed* matrix $S$ with the smallest $p$ that defines a partitioning of the columns of $A$ into $p$ groups.
3. Compute the numerical values of the entries of the *compressed* matrix $B \equiv AS$.
4. *Recover* the numerical values of the entries of $A$ from $B$.

The specific set of criteria used to define a seed matrix $S$—the partitioning problem—depends on whether the derivative matrix $A$ to be computed is a Jacobian (nonsymmetric) or a Hessian (symmetric). It also depends on whether the entries of the matrix $A$ are to be recovered from the compressed representation $B$ *directly* (without requiring any further arithmetic) or *indirectly* (for example, by solving for unknowns via successive substitutions). In previous works, we had provided a comprehensive review of graph coloring models that capture the partitioning problems in the various computational scenarios and developed novel algorithms for the coloring models [6, 8]. The efficacy of the coloring techniques in the overall process of Hessian computation via AD had been demonstrated in [7]. Implementations in C++ of all our coloring and related algorithms for Jacobian and Hessian computation have been assembled in a package called COLPACK [9]. Currently COLPACK is being interfaced with ADOL-C, which is an operator overloading based tool for the differentiation of functions specified in C/C++ [12].

In this paper, using a model from a chromatographic separation process as a test case and ADOL-C as an AD tool, we demonstrate the efficacy of the scheme in Algorithm 1 in computing a sparse Jacobian via a direct recovery method. In Section 2 we discuss an efficient implementation of a technique for sparsity pattern detection based on propagation of *index domains* that we have incorporated into ADOL-C and used in the first step of Algorithm 1. In Section 3 we discuss the distance-2 coloring algorithm we used in the second step to generate a seed matrix. To compute the compressed Jacobian in the third step, we used the vector forward mode of ADOL-C. We discuss Simulated Moving Beds, the context in which the Jacobians considered in our experiments arise, in Section 4 and present the experimental results in Section 5. The experimental results show that sparsity exploitation via coloring enables one to affordably compute Jacobians of dimensions that could not have been computed otherwise due to excessive memory or runtime requirements. The results also show that the index domain-based sparsity detection technique now available in ADOL-C is several orders of magnitude faster than the *bit vector*-based technique used earlier.

## 2 Automatic differentiation and sparsity pattern detection

AD provides exact derivative information about a smooth function $F : \mathbb{R}^n \to \mathbb{R}^m$, $x \mapsto F(x)$, given as a computer program by breaking down the computation of $F$ into a sequence of elementary evaluations upon which the chain rule of calculus is systematically applied. The decomposition of the function $F$ into its elementary components can be formalized as shown in Algorithm 2. There the *precedence relation* $j \prec i$ denotes that variable $v_i$ directly depends on variable $v_j$. The derivative of each elementary function $\varphi_i(v_j)_{j \prec i}$ with respect to its arguments $v_j$, $j \prec i$, is obtained easily, by a call to a library function. Then the chain rule is applied to the overall decomposition to obtain the derivatives of the function $F$ with respect to the input variables $x \in \mathbb{R}^n$. Depending on the starting point of this process—either at the beginning or at the

**Algorithm 2** Decomposition of function evaluation into elementary components.

**procedure** FUNCTION EVALUATION($y = F(x)$)
  **for** $i = 1$ **to** $n$ **do** : $v_{i-n} \leftarrow x_i$ ▷ *independent* variables
  **for** $i = 1$ **to** $l$ **do** : $v_i \leftarrow \varphi_i(v_j)_{j \prec i}$ ▷ *intermediate* variables
  **for** $i = 1$ **to** $m$ **do** : $y_i \leftarrow v_{l-i+1}$ ▷ *dependent* variables

end of the respective chain of computational steps—one gets the *forward* or the *reverse* mode of AD. The forward mode propagates derivatives from independent to dependent variables, and the reverse mode propagates derivatives from dependent to independent variables.

Under the framework followed in this paper, the task of making the computation of sparse derivative matrices via AD efficient begins with sparsity pattern detection. Several techniques for sparsity pattern detection have been suggested in previous studies for both of the major AD implementation paradigms, *source transformation* and *operator overloading*. The techniques could be classified as *static* and *dynamic*, depending on whether analysis is performed at compile time or run time. An example of a static technique in the context of source transformation is available in [16]. For dynamic techniques, two major approaches could be identified in the literature: *sparse vector*-based and *bit vector*-based. As exemplified by the SparsLinC library [2], a module in the source transformation AD tools ADIFOR and ADIC, a sparse vector based approach uses *sparse data structures*, instead of dense arrays, to execute a fundamental operation in AD codes, a (mathematical) linear combination of vectors. (Strictly speaking, the ADIFOR/SparsLinC combination is a mechanism for transparently exploiting sparsity in derivative computation; sparsity detection is a byproduct.) SparsLinC uses three different data structures to represent sparse vectors (one for vectors with at most one nonzero, a second for vectors with a few scattered nonzeros, and a third for vectors with a contiguous block of nonzeros) and heuristically switches from one representation to another as needed to reduce the large runtime overhead observed in earlier sparse vector based approaches [1].

Bit vector based approaches avoid the need for dynamic memory management, at the cost of increased memory requirement. When bit vectors are used, say, in the forward mode, the Jacobian is multiplied from the left by $n$ bit vectors, where $n$ is the number of independent variables; each arithmetic operation in the forward sweeps then corresponds to a logical **OR**, yielding the overall sparsity pattern of the Jacobian. Since one Jacobian-vector product needs to be performed for each independent variable, the complexity of this approach is $O(n \cdot OPS(F))$, where $OPS(F)$ is the number of operations involved in the evaluation of the function $F$. This time complexity can be reduced via Bayesian probing [13]. In terms of memory, the bit vector approach in its simplest form requires $1/64$th of the space needed to store a Jacobian, assuming representation of doubles and integers needs 64 bits. Thus far ADOL-C had a Jacobian sparsity pattern detection capability based on the use of bit vectors in such a manner. Bit vectors have also been used in the source transformation AD tool TAF [10].

In this work we develop a technique that could be viewed as a variant of the sparse-vector approach that minimizes dynamic memory management cost in the context of operator overloading. The idea is to extend the basic operations and intrinsic functions such that they propagate *index domains* in addition to function values. In particular, with each variable $v_i$ computed during the function evaluation, an index domain $\mathscr{X}_i$ satisfying the following condition is associated; see [11] for details.

$$\left\{ 0 \le j \le n : \frac{\partial v_i}{\partial x_j} \not\equiv 0 \right\} \subseteq \mathscr{X}_i. \tag{1}$$

**Algorithm 3** Propagation of index domains.

**procedure** COMPUTEINDEXDOMAINS($\mathscr{X}_i$)
  **for** $i = 1$ **to** $n$ **do** : $\mathscr{X}_{i-n} \leftarrow \{i\}$ ▷ *independent* variables
  **for** $i = 1$ **to** $l$ **do** : $\mathscr{X}_i \leftarrow \bigcup_{j \prec i} \mathscr{X}_j$ ▷ *intermediate* variables
  **for** $i = 1$ **to** $m$ **do** : $\mathscr{X}_i \leftarrow \mathscr{X}_{l-i+1}$ ▷ *dependent* variables

Here equality holds as long as no degeneracy arises in the function evaluation. Once the index domains of the dependent variables are obtained, the sparsity pattern of the corresponding Jacobian is readily available.

Using the internal function representation generated via operator-overloading, the index domains can be computed at runtime using the simple method outlined in Algorithm 3. Note that if a proper subset relation occurs in (1), then Algorithm 3 would yield an overestimate for the sparsity pattern. This results in an increase in runtime and space but not in incorrect numerical results. In Algorithm 3, for each operation, only the entries of the index domains of the operands are involved in the set union operation. The number of entries of the new index domain is bounded above by the maximum number of nonzeros per row of the Jacobian, $\rho_{\max}$. Hence, the complexity of Algorithm 3 is $O(\rho_{\max} \cdot OPS(F))$; see [11] for details.

We have incorporated into ADOL-C an array-based implementation of Algorithm 3 that has the complexity just mentioned. In the implementation, an integer array of fixed size (that could potentially be increased during the course of the algorithm) is used for each intermediate variable. The first two entries of an array are reserved for storing the current number of elements of the index domain and the current size of the array, respectively. If a variable occurs on the left side of an assignment, the corresponding array is updated to reflect the change in the index domain. If the size of the array becomes no longer large enough to store all indices, then the array is reallocated at runtime to increase its size. Since sparse Jacobian matrices in practice have only few nonzero entries per row, the initial (default) size of the arrays needs to be set at a fairly small value, and reallocation at later stages is hardly needed; in this study, 20 was used as the initial value. In the event that the initial estimate on array size is not large enough, note that one need not recompile the entire ADOL-C code, as the array reallocation happens at runtime. Note also that the sparsity structure determined by ADOL-C through this technique is correct so long as the control flow does not change. If the control flow changes, ADOL-C alerts the user via an appropriate error message. Runtime comparisons between this array-based implementation of Algorithm 3 and the bit vector-based approach previously available in ADOL-C will be presented in Section 5.

## 3 Compression via coloring

Once the sparsity pattern is determined, a sparse Jacobian can be computed efficiently using the compression-decompression scheme outlined in Algorithm 1. Curtis, Powell, and Reid [4] were the first to observe that a *structurally orthogonal* partition of a Jacobian matrix $A$—a partition of the columns of $A$ in which no two columns in a group share a nonzero at the same row index—gives a seed matrix $S$ where the entries of $A$ can be *directly* recovered from the compressed representation $B \equiv AS$. Coleman and Moré [3] modeled the associated problem of partitioning the columns of the Jacobian into the fewest possible groups as a distance-1 coloring problem on its *column intersection graph*.

As we have shown in [6], a structurally orthogonal partition of a Jacobian can equivalently, but more conveniently, be modeled as a partial distance-2 coloring on the bipartite

**Algorithm 4** A greedy partial distance-2 coloring algorithm.

**procedure** GREEDYPARTIALD2COLORING($G_b = (V_1, V_2, E)$)
  Let $u_1, u_2, \ldots, u_n$ be a given *ordering* of $V_2$, where $n = |V_2|$
  Initialize forbiddenColors with some value $a \notin V_2$
  **for** $i = 1$ **to** $n$ **do**
    **for each** vertex $w$ such that $(u_i, w) \in E$ **do**
      **for each** colored vertex $x$ such that $(w, x) \in E$ **do**
        forbiddenColors[color[$x$]] $\leftarrow u_i$
    color[$u_i$] $\leftarrow \min\{c > 0 : \text{forbiddenColors}[c] \neq u_i\}$

graph representation of the structure of the Jacobian. The *bipartite graph* $G_b(A) = (V_1, V_2, E)$ of a Jacobian matrix $A$ is a graph in which the vertex set $V_1$ corresponds to the rows of $A$, the set $V_2$ corresponds to the columns of $A$, and an edge joining a row vertex $r_i$ and a column vertex $c_j$ exists whenever the matrix element $a_{ij}$ is nonzero. A *partial distance-2* coloring of the graph $G_b$ on the vertex set $V_2$ is an assignment of colors (positive integers) to vertices in $V_2$ such that every pair of vertices from $V_2$ at a distance of exactly 2 edges from each other receives distinct colors. Clearly, two column vertices that receive the same color in a partial distance-2 coloring are at a distance greater than two edges from each other, and hence are structurally orthogonal. Thus, a partial distance-2 coloring is a partitioning of the columns of the matrix into groups of structurally orthogonal columns. In contrast to the column intersection graph, which has size proportional to the number of nonzeros in $A^T A$, the size of the bipartite graph of a Jacobian $A$ is proportional to the number of nonzeros in $A$. Primarily for this reason, the partial distance-2 coloring formulation uses less storage space and runtime compared to a distance-1 coloring formulation [6].

Finding a partial distance-2 coloring with the *fewest* colors is known to be an NP-hard problem. In this work, we used GREEDYPARTIALD2COLORING (outlined in Algorithm 4 and discussed in detail in [6, Section 3]) to find an approximate solution. The complexity of GREEDYPARTIALD2COLORING is $O(|E| \cdot \Delta(V_1))$, where $\Delta(V_1)$ is the maximum degree in the row vertex set $V_1$ of the input bipartite graph $G_b(A) = (V_1, V_2, E)$. Note that, $\Delta(V_1)$, which is the same as the maximum number of nonzeros per row $\rho_{\max}$ in the underlying Jacobian $A$, is a lower bound on the optimal number of colors needed.

## 4 The Simulated Moving Bed process

As a case study for evaluating the performance of the sparsity detection and coloring techniques discussed in the previous two sections in the context of Algorithm 1, we conducted experiments on Jacobians that arise in a model for liquid chromatographic separation. We review this model in the current section.

Liquid chromatographic separation is used in many chemical industrial processes as an efficient purification technique, since thermal methods such as distillation cannot be used for thermally unstable products or those with high boiling points. In liquid chromatography, a feed mixture is injected into one end of a column packed with adsorbent particles, and then pushed toward the other end with a desorbent (such as water or organic solvent). The mixture is separated by making use of the differences in the migration speeds of components in the liquid. In True Moving Bed (TMB) chromatography, the adsorbent moves in a counter-current direction to the liquid in a column. Since the transport of the adsorbent causes difficulties (such

as axial mixing of components), Simulated Moving Bed (SMB) chromatography, a pseudo counter-current process that mimics the operation of a TMB process, is used instead [14].

An SMB unit consists of several columns connected in a series. Fig. 1 shows a simplified model of an SMB unit with six columns, arranged in four zones, each of which consists of $N_{\text{dis}}$ compartments. In the figure, feed mixture and desorbent are supplied continuously to the SMB unit at inlet ports, while two products, extract and raffinate, are withdrawn continuously at outlet ports. The four streams, feed, desorbent, extract, and raffinate, are switched periodically to adjacent inlet/outlet ports, and rotate around the unit. Due to this cyclic operation, SMB never reaches a steady state, but only a Cyclic Steady State (CSS), where the concentration profiles at the beginning and at the end of a cycle are identical. Several different goals could be identified in an SMB process, maximizing throughput being a typical one. This objective is modeled mathematically as an optimization problem with constraints given by partial differential algebraic equations (PDAEs).

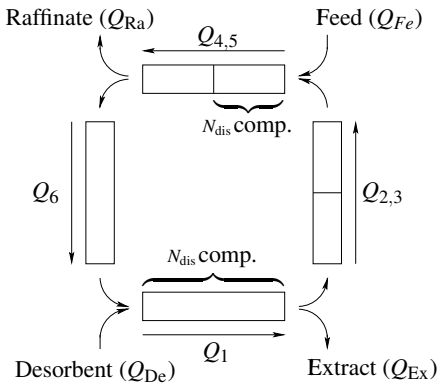


**Fig. 1**: A simple model of an SMB unit.

Numerical solution of the PDAEs requires efficient discretization and integration techniques. A straightforward approach here is to integrate the model until it reaches the CSS, update the operating parameters and repeat until the optimal values are found. To reduce the computational effort associated with the calculation of the CSS, approaches tailored for cyclic adsorption processes, where concentration profiles are treated as decision variables, have been developed. These approaches can be divided into two classes: those that discretize PDAEs only in space (single discretization) and those that discretize both in space and time (full discretization). Single discretization is well suited for complicated SMB processes, since it allows for the use of sophisticated numerical integration schemes. It results in comparatively small but dense derivative matrices [5, 18]. Full discretization is the method of choice if the step-size of the numerical integration can be fixed at a reasonable value [15]. The derivative matrices involved in the use of full discretization are typically sparse. We consider the computation of a sparse Jacobian for such a purpose. We use a standard collocation method for the full discretization of the state equation with nonlinear isotherms. The objective we have considered here is maximizing the feed throughput, which is achieved by finding optimal values for the four flow parameters $Q_1$, $Q_{\text{De}}$, $Q_{\text{Ex}}$, and $Q_{\text{Fe}}$ (see Fig. 1) and the duration $T$ of a cycle.

## 5 Experimental results

We considered ten Jacobians of varying sizes in our experiments. Table 1 lists the number of rows ($m$), columns ($n$), and nonzeros ($nnz$) in each Jacobian as well as the maximum, minimum, and average number of nonzeros per column ($\kappa$). The maximum, minimum, and average number of nonzeros per row in *every* problem instance are $\rho_{\max} = 6$, $\rho_{\min} = 2$, and $\bar{\rho} = 5.0$. The last column of Table 1 shows the number of colors $p$ used by the two partial distance-2 coloring algorithms we experimented with—the implementation of Algorithm 4

available in COLPACK and an implementation of a similar algorithm previously available in ADOL-C. In both of these greedy algorithms, the natural ordering of the vertices was used since it gave fewer colors compared to other ordering techniques.

The right part of Fig. 2 depicts the sparsity pattern of the smallest matrix in our collection (P1). The remaining nine instances have similar, but appropriately enlarged structures. As can be deduced from column $\kappa_{\max}$ of Table 1, there is a column in each of our test problems that contains nearly 10% of all the nonzeros in the matrix and is itself nearly 50% filled with nonzeros. This column, which is the fifth in each matrix, corresponds to the integration time (parameter $T$) of the system in the SMB model. Since the structure of this column and its neighborhood is hardly visible in the main plot at the right in Fig. 2, we have included the plot at the left where one "zooms" in the first eight columns.

**Table 1** Matrix statistics and number of colors used by a greedy algorithm (last column).

| P | $m$ | $n$ | $nnz$ | $\kappa_{\max}$ | $\kappa_{\min}$ | $\bar{\kappa}$ | $p$ |
|---|---|---|---|---|---|---|---|
| 1 | 4,370 | 4,380 | 24,120 | 2,375 | 1 | 5.0 | 8 |
| 2 | 8,570 | 8,580 | 47,340 | 4,655 | 1 | 5.0 | 8 |
| 3 | 17,145 | 17,155 | 95,670 | 9,555 | 1 | 5.0 | 8 |
| 4 | 25,545 | 25,555 | 142,590 | 14,235 | 1 | 5.0 | 8 |
| 5 | 50,745 | 50,755 | 283,350 | 28,275 | 1 | 5.0 | 8 |
| 6 | 76,115 | 76,125 | 426,470 | 42,775 | 1 | 5.0 | 8 |
| 7 | 101,495 | 101,505 | 569,600 | 57,275 | 1 | 5.0 | 8 |
| 8 | 152,245 | 152,255 | 855,860 | 86,275 | 1 | 5.0 | 8 |
| 9 | 270,195 | 270,205 | 1,520,360 | 153,435 | 1 | 5.0 | 8 |
| 10 | 506,095 | 506,105 | 2,850,320 | 287,995 | 1 | 5.0 | 8 |

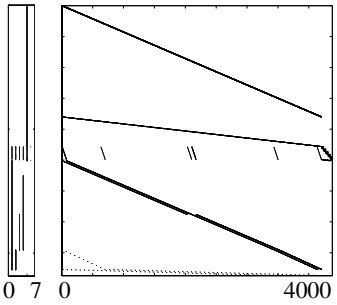


**Fig. 2** Sparsity pattern of problem P1.

Table 2 shows run times in seconds of various phases: the evaluation of the function $F$ being differentiated (eval($F$)) and the four steps S1, S2, S3, and S4 (see Algorithm 1) involved in the computation of the Jacobian using the vector forward mode of ADOL-C. The experiments were conducted on a Fedora Linux system with an AMD Athlon XP 1666 Mhz processor and 512 MB RAM. The gcc 4.1.1 compiler was used with -02 optimization.

For the sparsity detection step S1, results for both the new approach (propagation of index domains) and the previous approach used in ADOL-C (bit vectors) are reported in Table 2. For the coloring step S2, results for the routines from COLPACK and ADOL-C are reported. Both of these routines implement Algorithm 4, but their implementations and the data structures used to represent graphs differ. COLPACK uses the Compressed Storage Format, which consists of two integer arrays, one corresponding to vertices and the other to edges, to represent a graph. ADOL-C on the other hand uses *linked structures* to store a graph. The last two columns in Table 2 show the times spent in building these graph data structures by reading files specifying sparsity patterns from disk.

Figure 3 summarizes the trends suggested by the data in Table 2 (excluding the graph build routines) in the cases where the new sparsity detection method and the coloring functionality of COLPACK are used for steps S1 and S2, respectively. We make the following observations from the experimental results. Our observations involve comparisons with time complexities that were discussed in Sections 3 and 4. To ease reference here, we provide a summary of the complexities in Table 3.

*Function evaluation.* As expected, the runtime of function evaluation grows linearly with problem size (see left part of Figure 3).

| | | S1 | | S2 | | | | | graph build | |
|---|---|---|---|---|---|---|---|---|---|---|
| P | eval($F$) | old | new | ADO | COL | S3 | S4 | total | ADO | COL |
| 1 | 0.0001 | 0.4 | 0.016 | 0.007 | 0.004 | 0.0044 | 0.0009 | 0.0270 | 0.4 | 0.1 |
| 2 | 0.0002 | 1.2 | 0.018 | 0.006 | 0.004 | 0.0096 | 0.0017 | 0.0333 | 0.4 | 0.1 |
| 3 | 0.0004 | 3.6 | 0.038 | 0.014 | 0.009 | 0.0188 | 0.0033 | 0.0774 | 1.5 | 0.2 |
| 4 | 0.0007 | 7.8 | 0.062 | 0.020 | 0.013 | 0.0276 | 0.0050 | 0.1144 | 8.0 | 0.3 |
| 5 | 0.0017 | 29.4 | 0.104 | 0.040 | 0.026 | 0.0526 | 0.0095 | 0.1926 | 45.5 | 0.8 |
| 6 | 0.0032 | 67.4 | 0.159 | 0.060 | 0.040 | 0.0828 | 0.0151 | 0.2971 | 110.3 | 1.0 |
| 7 | 0.0045 | 122.3 | 0.238 | 0.082 | 0.053 | 0.1078 | 0.0205 | 0.4191 | 202.5 | 1.8 |
| 8 | 0.0063 | 275.7 | 0.332 | 0.121 | 0.080 | 0.1659 | 0.0309 | 0.6088 | 469.4 | 2.4 |
| 9 | 0.0108 | 870.7 | 0.580 | 0.233 | 0.142 | 0.2732 | 0.0435 | 1.0394 | 1,496.7 | 4.4 |
| 10 | 0.0199 | 3,050.9 | 1.072 | 0.416 | 0.266 | 0.5038 | 0.0834 | 1.9252 | 5,282.2 | 7.8 |

**Table 2.** Time in seconds spent on function evaluation and on the steps S1, S2, S3, and S4. The column *total* lists the sum (S1-new + S2-COL + S3 + S4). The last two columns list time spent on reading files from disk and building the bipartite graph data structures.

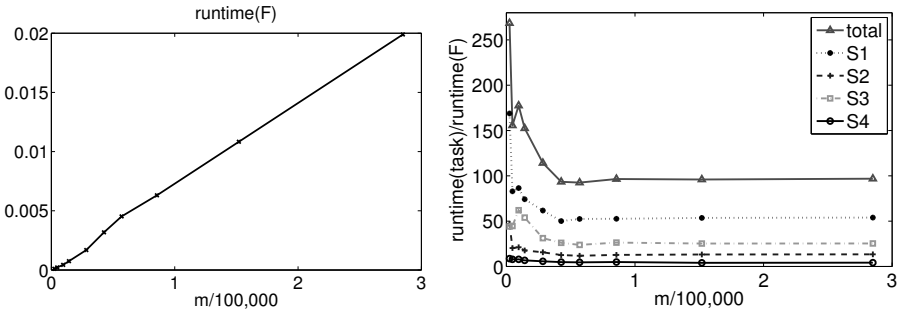


**Fig. 3.** Plots of time required for function evaluation (left) and time spent on the various steps normalized by the time for function evaluation (right).

| | S1 | | | | |
|---|---|---|---|---|---|
| eval($F$) | old | new | S2 | S3 | S4 |
| $O(nnz(\nabla F))$ | $O(n \cdot OPS(F))$ | $O(\rho_{\max} \cdot OPS(F))$ | $O(\rho_{\max} \cdot nnz(\nabla F))$ | $O(p \cdot OPS(F))$ | $O(nnz(\nabla F))$ |

**Table 3.** Summary of time complexity results.

*S1: Sparsity pattern detection.* Table 2 shows that the approach based on propagation of index domains is several orders of magnitude faster than the approach based on bit vectors. This agrees well with the complexities given in Table 3. Focusing on the former, as the right part in Fig. 3 shows, the runtime for sparsity detection normalized relative to runtime for function evaluation remained constant as problem size increased. This agrees well with the theoretical result derived in [11, Section 6.1], where the operation count for the determination

of sparsity pattern is bounded by $\gamma \cdot \rho_{\max} \cdot OPS(F)$, where $\gamma$ is a small constant. Since $\rho_{\max}$ equals six in each of our test cases, one can deduce from Fig. 3 that $\gamma$ is only around nine.

*S2: Coloring and generation of seed matrix.* The coloring algorithms invariably used just eight colors for all the problem sizes considered. These results are either optimal or at most only two colors off the optimal, since the lower bound in each case, $\rho_{\max}$, is six. This is an impressive result, since Fig. 2 shows that the sparsity patterns of the Jacobians is fairly complex when compared with Jacobians of structured grids. In terms of runtime, the observed results for the COLPACK function completely agree with the complexity given in Table 3. As can be seen from Table 2, the coloring routine previously available in ADOL-C is somewhat slower than the COLPACK routine. The big difference between the two, however, lies in the time needed to build the graph data structures: the last two columns of Table 2 show that the ADOL-C routine is up to three orders of magnitude slower than the COLPACK routine.

*S3: Computation of the compressed Jacobian.* Theoretically, the ratio between the number of operations involved in the evaluation of a compressed Jacobian with $p$ columns and the number of operations involved in the evaluation of the function itself is expected to be bounded by $1 + 1.5p$; see [11, Section 3.2] for details. Since $p$ equals eight for each of our test problems, the expected (constant) bound is 13. As can be seen from Fig. 3, the observed ratio is indeed a constant and at most only twice as much; for the larger problems it is about 25. But, considering the large sizes of these problems and considering that memory access times are not accounted for, the deviation from the theoretical value is minimal.

*S4: Recovery of original Jacobian entries.* The recovery step involves a straightforward row-by-row mapping of nonzero entries from the compressed to the original Jacobian. The observed run time behavior reflects this fact.

*Total runtime.* As one can see from Fig. 3, for the problems we experimented with, the ratio between the *total* runtime needed to compute a sparse Jacobian and the runtime for the function evaluation is observed to be a constant around 100. This fairly small number shows that the sparse approach is effective for large-scale Jacobian computations. The alternative "dense" approach is not feasible at all: out of the ten Jacobians in our test bed, only the smallest could be computed without exploiting sparsity due to excessive memory requirement. It is interesting to note that the multiplicative factor 100 observed in our experiments is distributed among the four involved steps in the ratio 55 : 25 : 15 : 5 for the steps S1 : S3 : S2 : S4, respectively. This shows that the coloring (S2) and recovery (S4) steps are by far the cheapest.

## 6 Conclusion

We demonstrated that automatic differentiation implemented via operator overloading together with efficient coloring algorithms constitute a powerful approach for computing sparse Jacobian matrices. One of the contributions of this work is an efficient implementation of a sparsity detection technique based on propagation of index domains. The approach can be extended to detect sparsity pattern of Hessians [17]. We also plan to develop similar sparsity detection techniques for derivative matrices in the context of source transformation based AD tools being developed within the framework of OpenAD.

**Acknowledgments.** We thank Paul Hovland for helpful discussions on sparsity detection in AD and Christian Bischof and the anonymous referees for their valuable comments on earlier versions of this paper. This work is supported in part by the U.S. Department of Energy through grant DE-FC-0206-ER-25774 awarded to the CSCAPES Institute and by the U.S. National Science Foundation through grant CCF-0515218.

## References


1. Bartholomew-Biggs, M.C.B.B.L., Christianson, B.: Optimization and automatic differentiation in ADA. Optimization Methods and Software **4**, 47–73 (1994)
2. Bischof, C.H., Khademi, P.M., Bouaricha, A., Carle, A.: Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. Optimization Methods and Software **7**, 1–39 (1996)
3. Coleman, T., Moré, J.: Estimation of sparse Hessian matrices and graph coloring problems. Math. Program. **28**, 243–270 (1984)
4. Curtis, A., Powell, M., Reid, J.: On the estimation of sparse Jacobian matrices. J. Inst. Math. Appl. **13**, 117–119 (1974)
5. Diehl, M., Walther, A.: A test problem for periodic optimal control algorithms. Tech. Rep. MATH-WR-01-2006, TU Dresden (2006)
6. Gebremedhin, A., Manne, F., Pothen, A.: What color is your Jacobian? Graph coloring for computing derivatives. SIAM Rev. **47**(4), 629–705 (2005)
7. Gebremedhin, A., Pothen, A., Tarafdar, A., Walther, A.: Efficient computation of sparse Hessians using coloring and automatic differentiation. INFORMS J. Comput. Under review, submitted in 2007.
8. Gebremedhin, A., Tarafdar, A., Manne, F., Pothen, A.: New acyclic and star coloring algorithms with application to computing Hessians. SIAM J. Sci. Comput. **29**, 1042–1072 (2007)
9. Gebremedhin, A., Tarafdar, A., Pothen, A.: COLPACK: A graph coloring package for supporting sparse derivative matrix computation (2008). In preparation.
10. Giering, R., Kaminski, T.: Automatic sparsity detection implemented as source-to-source transformation. In: Lecture Notes in Computer Science, vol. 3394, pp. 591–598 (1998)
11. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. No. 19 in Frontiers in Appl. Math. SIAM, Philadelphia (2000)
12. Griewank, A., Juedes, D., Utke, J.: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. ACM Trans. Math. Softw. **22**, 131–167 (1996)
13. Griewank, A., Mitev, C.: Detecting Jacobian sparsity patterns by Bayesian probing. Math. Program. Ser. A **93**, 1–25 (2002)
14. Kawajiri, Y., Biegler, L.: Large scale nonlinear optimization for asymmetric operation and design of Simulated Moving Beds. J. Chrom. A **1133**, 226–240 (2006)
15. Kawajiri, Y., Biegler, L.: Large scale optimization strategies for zone configuration of simulated moving beds. Tech. rep., Carnegie Mellon University (2007)
16. Tadjouddine, M., Faure, C., Eyssette, F.: Sparse Jacobian computation in automatic differentiation by static program analysis. In: Lecture Notes in Computer Science, vol. 1503, pp. 311–326 (1998)
17. Walther, A.: Computing sparse Hessians with automatic differentiation. ACM Trans. Math. Softw. **34**(1) (2008). Article No 3
18. Walther, A., Biegler, L.: A trust-region algorithm for nonlinear programming problems with dense constraint Jacobians. Tech. Rep. MATH-WR-01-2007, Technische Universität Dresden (2007). (submitted to Comput. Opt. Appl.)