Available online at www.sciencedirect.com
ScienceDirect
Procedia Computer Science 00 (2011) 1–10
Procedia Computer Science

# Sparse Jacobian Computation Using ADIC2 and ColPack

Sri Hari Krishna Narayanan, Boyana Norris, Paul Hovland

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA*

Duc C. Nguyen, Assefaw H. Gebremedhin

*Department of Computer Science, Purdue University, West Lafayette, IN, USA*

**Abstract**

Many scientific applications benefit from the accurate and efficient computation of derivatives. Automatically generating these derivative computations from an applications source code offers a competitive alternative to other approaches, such as less accurate numerical approximations or labor-intensive analytical implementations. ADIC2 is a source transformation tool for generating code for computing the derivatives (e.g., Jacobian or Hessian) of a function given the C or C++ implementation of that function. Often the Jacobian or Hessian is sparse and presents the opportunity to greatly reduce storage and computational requirements in the automatically generated derivative computation. ColPack is a tool that compresses structurally independent columns of the Jacobian and Hessian matrices through graph coloring approaches. In this paper, we describe the integration of ColPack coloring capabilities into ADIC2, enabling accurate and efficient sparse Jacobian computations. We present performance results for a case study of a simulated moving bed chromatography application. Overall, the computation of the Jacobian by integrating ADIC2 and ColPack is approximately 40% faster than a comparable implementation that integrates ADOL-C and ColPack when the Jacobian is computed multiple times.

*Keywords:* automatic differentiation, ADIC2, sparse derivative computation, ColPack
*PACS:* 02.60.Cb, 02.60.Jh, 02.70.Wz
*2010 MSC:* 68N99

## 1. Introduction

Derivatives play an important role in scientific applications and other areas, including numerical optimizations, solution of nonlinear partial differential equations, and sensitivity analysis. Automatic differentiation (AD) is a family of techniques for computing derivatives given a

*Email addresses:* snarayan@mcs.anl.gov (Sri Hari Krishna Narayanan), norris@mcs.anl.gov (Boyana Norris), hovland@mcs.anl.gov (Paul Hovland), nguyend@purdue.edu (Duc C. Nguyen), agebreme@purdue.edu (Assefaw H. Gebremedhin)

program that computes some mathematical function. In general, given a code $C$ that computes a function $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$ with $n$ inputs and $m$ outputs, an AD tool produces code $C'$ that computes $f' = \partial y/\partial x$, or the derivatives of some of the outputs $y$ with respect to some of the inputs $x$. We call $x$ the *independent variable* and $y$ the *dependent variable* and denote the Jacobian matrix $f'(x)$ by $J$. Other quantities, such as Jacobian-vector products, can also be computed through AD without explicitly forming $J$. The basic concepts of AD were introduced in 1950 [1, p. 12], and the capabilities and popularity of AD tools have been growing over the past couple of decades.

In many cases the Jacobian (or Hessian) being computed is sparse and can be compressed to avoid storing and computing with zeros. Curtis, Powell, and Reid demonstrated that when two or more columns of a Jacobian are structurally orthogonal (that is, there is no row in which more than one column has a nonzero), they can be approximated simultaneously using finite differences by perturbing the corresponding independent variables simultaneously [2]. Coleman and Moré showed that the problem of identifying structurally orthogonal Jacobian columns can be modeled as a graph coloring problem [3]. The methods developed for finite-difference approximations are readily adapted to automatic differentiation with appropriate initialization of the seed matrix [4]. Exploiting sparsity while using AD can also yield better performance than finite-difference (FD) approximations because AD computes the entire (compressed) Jacobian simultaneously, whereas FD computes it one (compressed) column at a time.

### *1.1. Framework for Sparse Computation*

Given a function $f$ whose $(m \times n)$ derivative matrix $J$ is sparse, the framework we employ to efficiently compute the matrix $J$ using AD involves the following four steps:

1. Determine the *sparsity pattern* of the matrix $J$.
2. Using a *coloring* on an appropriate graph of $J$, obtain an $n \times p$ *seed* matrix $S$ with the smallest $p$ that defines a partitioning of the columns of $J$ into $p$ groups.
3. Compute the numerical values in the *compressed* matrix $B \equiv JS$.
4. *Recover* the numerical values of the entries of $J$ from $B$.

The first and third steps of this scheme are necessarily carried out by an AD tool, whereas the second and fourth steps could be performed by a separate, independent tool. This separation of concerns offers an opportunity for a tool developed for the second and fourth steps to be interfaced with any AD tool.

### *1.2. ColPack*

ColPack [5] is a software package that comprises implementations of various algorithms for graph coloring and recovery, that is, the second and fourth steps. The coloring models used come in several variations depending on whether the derivative matrix to be computed is a Jacobian (nonsymmetric) or a Hessian (symmetric) and whether the derivative matrix is compressed such that the nonzero entries are to be recovered directly (with no additional arithmetic work) or indirectly (by substitution). Table 1 gives an overview of the coloring models used in ColPack for sparse Jacobian and Hessian computations. Figure 1 illustrates how a partitioning of the columns of a Jacobian into structurally orthogonal groups is modeled as a (partial) distance-2 coloring of the bipartite graph of the Jacobian.

Every problem listed in Table 1 is NP-hard to solve optimally [6, 7]. The algorithms in ColPack for solving these problems are fast, and yet effective, greedy heuristics [5]. They are

Table 1: Overview of graph coloring models used in ColPack for computing sparse derivative matrices. The Jacobian is represented by its *bipartite* graph and the Hessian by its *adjacency* graph. NA stands for not applicable.

| **Matrix** | **Unidirectional Partition** | **Bidirectional Partition** | **Recovery** |
|---|---|---|---|
| Jacobian | *distance-2 coloring* | *star bicoloring* | Direct |
| Hessian | *star coloring* | NA | Direct |
| Jacobian | NA | *acyclic bicoloring* | Substitution |
| Hessian | *acyclic coloring* | NA | Substitution |

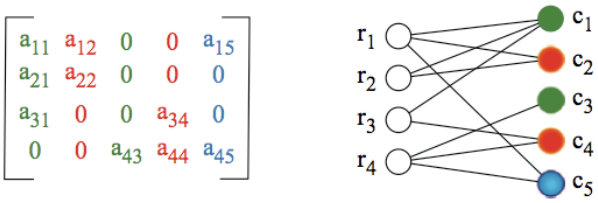


Figure 1: Partitioning of a matrix into structurally orthogonal groups and its representation as a (partial) distance-2 coloring on a bipartite graph (of the matrix).

greedy in the sense that vertices are colored sequentially one at a time and the color assigned to a vertex is never changed. The *order* in which vertices are processed in a greedy heuristic determines the number of colors used by the heuristic. ColPack contains implementations of various effective ordering techniques for each of the coloring problems it supports. ColPack is written in an object-oriented fashion in C++ using the Standard Template Library (STL). It is designed to be modular and extensible.

### *1.3. ADIC2*

The implementation of AD tools is normally based on one of two approaches: operator overloading (e.g., ADOL-C [8]) or source-to-source transformation (e.g., TAPENADE [9], TAF [10], TAC++ [11], OpenAD/F [12, 13], and ADIC2 [14][1]).

ADIC2 is a component-based source-to-source transformation AD tool for C and C++ [14]. It can handle both forward mode and reverse mode AD. It is based on the ROSE compiler framework [16, 17] and leverages several tools from the OpenAD project [13] as components. The AD process as implemented by ADIC2 is described in detail in [14]. Figure 2 shows a sample input and the output code generated by ADIC2. The type of each *active* variable[2] is changed to *DERIV_TYPE*, which is a C structure containing a scalar value and a dense array for storing the partial derivatives of each active variable w.r.t. each independent variable, as shown below.

```
typedef struct {
        double val;
        double grad[ADIC_GRADVEC_LENGTH];
} DERIV_TYPE;
```

[1] A complete survey of AD tools and implementation techniques is outside the scope of this article; more information is available at [15].

[2] *Active* variables lie on the computational path between independent and dependent variables.

The derivative code generated by ADIC2 is compiled and linked to a runtime library that provides implementations of functions (or macros) for manipulating *DERIV_TYPE* objects. Derivatives are propagated by applying the chain rule to combine partial derivatives, which, in the forward vector-gradient AD mode used in this example, involves summing different numbers of scalar-*grad* array products. For example, an *axpy2* operation $Y \leftarrow Y + \alpha_1 * X_1 + \alpha_2 * X_2$ will access each element of $X_1.grad$, $X_2.grad$, and *Y.grad* and update *Y.grad*. The size of the array (`ADIC_GRADVEC_LENGTH`) depends on the number of independent variables, and the number of operations required depends on the number of active variables.

**(a) Input code:**

```
void mini1(double *y, double *x, int n)
{
  int i;
  *y = 1.0;
  for (i = 0; i < n; i=i+1) {
    *y = *y * x[i];
  }
}
```

**(b) Generated forward-mode AD code:**

```
#include "ad_types.h"
void ad_mini1(DERIV_TYPE *y,DERIV_TYPE *x,int n)
{
  int ad_i;
  DERIV_val( *y) = 1.00000F;
  ADIC_ZeroDeriv(DERIV_TYPE_ref( *y));
  for (ad_i = 0; ad_i < n; ad_i = (ad_i + 1)) {
    DERIV_TYPE ad_TempVarprp_1;
    DERIV_TYPE ad_TempVarprp_0;
    double ad_TempVarlin_1;
    double ad_TempVarlin_0;
    double ad_TempVardly_0;
    ad_TempVardly_0 = DERIV_val( *y) * DERIV_val(x[ad_i]);
    ad_TempVarlin_0 = DERIV_val(x[ad_i]);
    ad_TempVarlin_1 = DERIV_val( *y);
    DERIV_val( *y) = ad_TempVardly_0;
    ADIC_SetDeriv(DERIV_TYPE_ref( *y),DERIV_TYPE_ref(ad_TempVarprp_0));
    ADIC_SetDeriv(DERIV_TYPE_ref(x[ad_i]),DERIV_TYPE_ref(ad_TempVarprp_1));
    ADIC_Sax_Dense1(ad_TempVarlin_0,DERIV_TYPE_ref(ad_TempVarprp_0),DERIV_TYPE_ref( *y));
    ADIC_Saxpy(ad_TempVarlin_1,DERIV_TYPE_ref(ad_TempVarprp_1),DERIV_TYPE_ref( *y));
  }
}
```

Figure 2: (a) Example input code; (b) generated forward-mode differentiated code.

### *1.4. Contributions*

In this work, we describe new automated sparsity detection (ASD) capabilities we have added to ADIC2 by implementing a new version of the SparsLinC library. We also have interfaced ADIC2 with ColPack to enable sparse derivative computation. We demonstrate the advantage of the combined capability using an application from chemical engineering. This is the first time ColPack has been been interfaced with a source-to-source AD tool.

The rest of the paper is organized as follows. In Section 2 we present an overview of the process of computing compressed Jacobians using ADIC2 and ColPack, and we briefly describe

the new version of SparsLinC. In Section 3 we describe the application we used to evaluate the new capability and we present experimental results. We discuss related work in Section 4 and conclude in Section 5 with a brief description of future work.

## 2. Integration Approach

The main steps involved in computing a Jacobian by using ADIC2 are listed in Table 2. Compressed Jacobian computations require several extra initialization steps and possibly more expensive Jacobian recovery from the compressed format. These costs are incurred only when the Jacobian structure changes and are thus normally amortized by the greatly reduced cost of computing multiple Jacobians.

Table 2: Steps required for computing a dense (left) and a compressed sparse (right) Jacobian using ADIC2 and ColPack.

1. Initialization
   - Specify independent and dependent variables (create identity seed matrix for full Jacobian).
2. Compute derivatives.
3. Extract derivatives for use in later computations (simple copy).

1. Initialization
   - Specify independent and dependent variables.
   - Compute sparsity pattern (SparsLinC).
   - Construct graph and perform coloring (ColPack).
   - Create compressed seed matrix.
2. Compute derivatives.
3. Extract derivatives for use in later computations.
   - Recover from compressed format (ColPack)

*Sparsity detection.* Sparsity detection techniques in AD can be classified as *static* or *dynamic*, depending on whether analysis is performed at compile time or runtime. For an example of a static technique in the context of a source transformation–based AD tool, see [18]. For dynamic techniques, two major approaches can be identified: sparse vector–based and bit vector–based. The sparse vector–based approach has the advantage over bit vector–based approaches in that it requires less memory. But it is potentially slower because it involves dynamic manipulation of sparse data structures. We have initially adopted the sparse vector–based approach and implemented it in the SparsLinC library. Previous versions of the SparsLinC library have been used by ADIFOR [19] and ADIC1 [20] to support sparse dynamic storage of derivatives. We reimplemented SparsLinC completely for ADIC2, enabling both runtime sparsity detection (without derivative computations) and sparse vector–based derivative computations. Internally, SparsLinC defines a data structure that consists of a set of integers called the *index set* for each *active variable*. Entries in the set are the indices of the nonzero elements within the dense gradient array of the original *DERIV_TYPE* if *DERIV_TYPE* is used. In the ASD version of SparsLinC, the ADIC2-generated functions for operations on dense arrays were rewritten to instead insert or remove elements of the index set. For example an *axpy2* operation $Y \leftarrow Y + \alpha_1 * X_1 + \alpha_2 * X_2$ is implemented in SparsLinC to insert the union of the index sets of $X_1$ and $X_2$ respectively into the index set of $y$. Running ADIC2 with the SparsLinC library results in a data structure containing the sparsity structure of the Jacobian represented as sets of nonzero elements.

*Coloring and seed matrix generation.* The sparsity pattern produced by ADIC2 and SparsLinC serves as an input to ColPack. The input is used by ColPack to construct a suitable graph, compute an appropriate coloring, and, using the coloring, obtain a Jacobian seed matrix. Internally,

the compressed Jacobian is stored in statically allocated dense arrays (the size of each array is equal to the number of colors). ColPack provides the nonzero entries of the original (uncompressed) Jacobian through its recovery routines.

## 3. Experimental Evaluation

### 3.1. Example Application

Liquid chromatography is a frequently used purification technique in the chemical industry to separate products that are thermally unstable or have high boiling points, where distillation is inapplicable. 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 an organic solvent). The mixture is separated by making use of the differences in the migration speeds of components in the liquid. Simulated moving bed (SMB) chromatography is a technique used to mimic true moving bed (TMB) chromatography, where the adsorbent moves in a countercurrent direction to the liquid in a column [21].

An SMB unit consists of several columns connected in a series. Figure 3 shows a simplified model of an SMB unit with six columns, arranged in four zones, each of which consists of $N_{\text{dis}}$ compartments. 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. Because of this cyclic operation, SMB never reaches a steady state, but only a cyclic steady state, where the concentration profiles at the beginning and at the end of a cycle are identical.

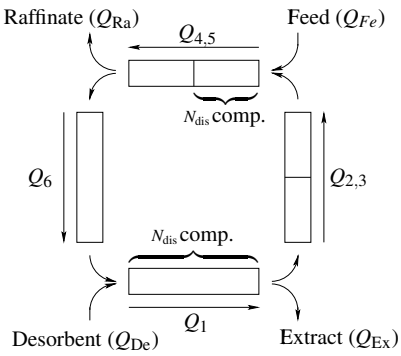


Figure 3: Simple model of an SMB unit.

Maximizing throughput is a common goal associated with an SMB process. This objective is modeled mathematically as an optimization problem with constraints given by partial differential algebraic equations (PDAEs). The PDAE-constrained optimization problem can be solved by employing various discretization and integration techniques [22, 23]. We target the case where an approach tailored for cyclic adsorption processes (where concentration profiles are treated as decision variables) is used, and the PDAEs are discretized both in space and time (full discretization). The derivative matrices involved in the use of full discretization are typically sparse. We focus in this work on the computation of a sparse Jacobian (of the constraint function) in the solution of the optimization problem modeling the SMB process.

### 3.2. Experimental Results

This section presents experimental results of computating the Jacobian of the constraint function in the SMB application. The resulting full Jacobian dimensions for the problem size we consider are $4570 \times 4580$. We compare the derivative computation performance of the ADOL-C operator overloading approach with the performance of the codes generated by different configurations of ADIC2. We measure 100 Jacobian evaluations because in a typical optimization algorithm (as well as other types of applications) the Jacobian matrix structure remains the same

Table 3: Wall-clock time for evaluating the Jacobian of the constraint function 100 times, and the ratio between the wall-clock time for computing a single $J$ (including a breakdown of overhead) using different AD approaches and the original function time (Row 1). The recurring costs for different AD techniques, represented by 100 $J$ evaluations, are highlighted in bold font. Unhighlighted lines represent one-time overheads.

| | ADOL-C | | ADIC2 | |
|---|---|---|---|---|
| | Time (sec) | Ratio | Time (sec) | Ratio |
| **1. Constraint function evaluation** | **0.000156** | **1** | **0.000156** | **1** |
| **2. Dense $J$ (total for 100, vector)** | **359.96** | **23,074.55** | **1195.72** | **76,649.03** |
| **3. Sparse $J$ (total for 100, SparsLinC)** | – | – | **19.34** | **1,239.57** |
| **4. Sparse $J$ (total for 100, AD Tool+ColPack)** | **1.8630** | **116.98** | **1.3263** | **77.56** |
| Sparse $J$ computation breakdown: | | | | |
| 4.1. Sparsity detection | 0.0260 | 166.57 | 0.0706 | 452.53 |
| 4.2. Seed matrix computation (total) | 0.0121 | 77.57 | 0.0488 | 312.74 |
| 4.2.1 Graph construction | 0.0097 | 62.37 | 0.0461 | 295.69 |
| 4.2.2 Graph coloring | 0.0019 | 11.83 | 0.0022 | 13.92 |
| 4.2.3 Seed collection | 0.0005 | 3.37 | 0.0005 | 3.13 |
| **4.3. Compressed $J$ (vector)** | **1.77** | **113.40** | **0.68** | **43.34** |
| **4.4. Recovery** | **0.055** | **3.58** | **0.53** | **34.03** |

for multiple evaluations of the Jacobian. The performance results are summarized in Table 3. The experiments were conducted on a four-processor server with AMD 8431 six-core 2.4 GHz processors with 256 GB of DDR2 667MHz RAM, running Linux kernel version 2.6.18 (x86_64). All measurements are for serial code.

The "Ratio" columns contain the costs for the different AD computation approaches and some of their constituent steps, normalized by the constraint function computation time shown in Row 1. Row 2 shows the performance of a 100 *full dense* Jacobian evaluations, without exploiting sparsity. In the ADIC2 case, this normally means that each active variable is associated with a 4580-element statically allocated array for storing the partial derivatives w.r.t. the $4,580$ independent variables. The constraint function implementation declares a large number of intermediate temporary arrays, which causes the ADIC2-generated code to overrun stack space when the differentiated function is called. Therefore, we used dynamic memory allocation for temporaries in the dense case shown in Row 2; this approach is slower than using static arrays, but is nevertheless the only feasible dense computation option for this code.

Rows 3 and 4 show the *total* times for 100 Jacobian evaluations using two principally different approaches: Row 3 uses sparse vectors to store only nonzero Jacobian values, while Row 4 uses the graph-coloring capabilities of ColPack to produce a compressed *dense* Jacobian representation with only 8 columns corresponding to the 8 colors determined during the coloring.

In the coloring-based approach, ADIC2 offers two choices for computing the compressed Jacobians while exploiting sparsity, which can be employed in Row 4.3 in Table 3: (A) dense scalar gradient (most similar in performance to using finite differences) with coloring, and (B) dense vector-gradient compressed $J$ computation using coloring (this is the version included in the table Row 4). The time for computing $J$ by using approach (A) is not included because (B) was 2 times faster, as can be expected since it employs array derivative accumulation operations rather than scalar operations.

Some of the ADIC-2 compression overhead costs (Rows 4.1 and 4.2) are higher than those for ADOL-C because of the limitations of the current sparse vector implementation in SparsLinC, which uses a C++ STL set to implement the index sets. We have not yet optimized the internal

representation because this is a new SparsLinC implementation. Because the sparsity detection mechanisms used in ADOL-C and ADIC2+SparsLinC are similar in principle, we should be able to achieve similar low overheads with future optimizations.

Row 4 shows a breakdown of the sparse computation into the four steps (sparsity detection, seed matrix computation, compressed Jacobian computation, and recovery) outlined in Section 1.1. The seed matrix computation step is further broken down into the three underlying substeps: construction of the graph used by ColPack from the internal representation of the sparsity pattern in the AD tool (Row 4.2.1), coloring of the constructed graph (Row 4.2.2), and seed matrix collection from the coloring (Row 4.2.3). The significant difference in graph construction times (Row 4.2.1) between ADOL-C and ADIC2 is caused by the differences in the underlying data structures used by each tool to represent sparsity patterns. The graph construction for the ADOL-C case (which uses compressed row format using simple arrays) is faster than the ADIC2 case (which uses STL data structures via the current implementation of SparsLinC, as described in Section 2).

Overall, the computation of the compressed ADIC2-generated vector-mode Jacobian is about 40% faster than the ADOL-C compressed Jacobian computation for multiple evaluations of $J$ despite the relatively higher overhead costs for the sparsity detection, seed matrix construction, and recovery steps in the current ADIC2-SparsLinC implementation.

## 4. Related Work

Jacobian or Hessian sparsity can be detected either at runtime or statically, or through a hybrid static/runtime approach. Runtime ASD is normally implemented through the propagation of bitvectors or similar structure containing the sparsity information [24]. A number of AD tools support runtime ASD (e.g., ADOL-C, ADIC version 1, and TAF). Our current approach is perhaps most similar to the sparsity detection approach in TAF [25], which transforms the original function computation into a code that propagates bitvectors and combines them by logical "or" operations. In our current implementation, we rely on STL sets instead of bitvectors and, at present generate only forward-mode sparsity detection code.

ColPack was interfaced with ADOL-C in previous related work [26]. In that work, ADOL-C acquired a sparsity pattern detection technique for Jacobians based on propagation of *index domains*. The sparsity detection capability previously available in ADOL-C was based on *bit vectors*. The detection technique based on index domains is a variant of the sparse-vector approach; the technique additionally strives to minimize dynamic memory management cost in the context of AD via operator overloading. Experiments carried out in [26] on Jacobian computation showed that the sparsity pattern detection step (based on index domains) was the most expensive of the four steps of the procedure for sparse derivative computation outlined in Section 1.1—it accounted for nearly 55% of the total runtime. When bit vectors were used, the detection step was even more expensive, in terms of both runtime and memory requirement. The idea of index domains propagation was extended to the detection of sparsity patterns of Hessians and implemented in ADOL-C in another work [27]. The capability was used together with ColPack to compute sparse Hessians arising in an optimal electric power flow problem [28].

The pioneering work on graph coloring software for sparse derivative computation was done by Coleman, Garbow, and Moré in the mid-1980s [29, 30]. They developed Fortran software packages for estimating Jacobians and Hessians by using finite differencing. ColPack is developed to support both AD and FD and is implemented in C++ with efficiency, modularity, and extendibility as design objectives; indeed for some computational scenarios (see Table 1), it uses

more accurate coloring models and algorithms than those used in [29, 30]. Recently, Hasan, Hossain, and Steihaug [31] presented preliminary work on a planned software toolkit for computing a Jacobian (using a direct method) when the sparsity pattern is known a priori.

## 5. Conclusions and Future Work

We demonstrated the advantages of exploiting sparsity in the computation of sparse Jacobians via source-transformation based AD using an optimization problem in chromatographic separation as a case study. Our approach involved the combined use of the newly redesigned AD tool ADIC2 and the software package ColPack, comprising graph coloring and related functionalities for sparse derivative computation.

We implemented automated sparsity detection using a new version of SparsLinC. We plan to optimize the performance of SparsLinC to reduce the overhead of the compression process by employing static analysis and also improving the implementation of the runtime library.

We provided a minimal interface between ADIC2 and ColPack sufficient for Jacobian computation by unidirectional compression. We plan to implement interfaces needed for Hessian computation and Jacobian computation by bidirectional compression, where both the forward and reverse modes of AD are employed.

We also plan to incorporate the compressed Jacobian capabilities into the PETSc numerical toolkit [32, 33, 34] by building on the existing PETSc-ADIC2 integration [35].

## Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contracts DE-AC02-06CH11357 and DE-FC-0206-ER-25774, and by the National Science Foundation grant CCF-0830645. We thank Lorenz T. Biegler and Andrea Walther for sharing their code for the SMB model with us.

## References

[1] A. Griewank, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, no. 19 in Frontiers in Appl. Math., SIAM, Philadelphia, PA, 2000.

[2] A. R. Curtis, M. J. D. Powell, J. K. Reid, On the estimation of sparse Jacobian matrices, J. Inst. Math. Appl. 13 (1974) 117–119.

[3] T. F. Coleman, J. J. Moré, Estimation of sparse Jacobian matrices and graph coloring problems, SIAM J. Numer. Anal. 20 (1) (1983) 187–209.

[4] B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, A. Griewank, Computing large sparse Jacobian matrices using automatic differentiation, SIAM J. Sci. Comput. 15 (2) (1994) 285–294.

[5] A. Gebremedhin, ColPack Web Page, `http://www.cscapes.org/coloringpage/`.

[6] A. Gebremedhin, F. Manne, A. Pothen, What color is your Jacobian? Graph coloring for computing derivatives, SIAM Review 47 (4) (2005) 629–705.

[7] A. Gebremedhin, A. Tarafdar, F. Manne, A. Pothen, New acyclic and star coloring algorithms with applications to Hessian computation, SIAM J. Sci. Comput. 29 (2007) 1042–1072.

[8] A. Griewank, D. Juedes, H. Mitev, J. Utke, O. Vogel, A. Walther, ADOL-C: A package for the automatic differentiation of algorithms written in C/C++, Technical Report, Technical University of Dresden, Institute of Scientific Computing and Institute of Geometry (1999).

[9] V. Pascual, L. Hascoët, TAPENADE for C, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann (Eds.), Advances in Automatic Differentiation, Vol. 64 of Lecture Notes in Computational Science and Engineering, 2008, pp. 199–210.

[10] R. Giering, T. Kaminski, Recipes for adjoint code construction, ACM Trans. Math. Software 24 (4) (1998) 437–474.

[11] M. Voßbeck, R. Giering, T. Kaminski, Development and first applications of TAC++, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 187–197. doi:10.1007/978-3-540-68942-3_17.

[12] J. Utke, U. Naumann, M. Fagan, N. Tallent, M. Strout, P. Heimbach, C. Hill, C. Wunsch, OpenAD/F: A modular, open-source tool for automatic differentiation of Fortran codes, ACM Transactions on Mathematical Software 34 (4) (2008) 18:1–18:36. doi:10.1145/1377596.1377598.

[13] OpenAD Web Page, `http://www.mcs.anl.gov/OpenAD/`.

[14] S. H. K. Narayanan, B. Norris, B. Winnicka, ADIC2: Development of a component source transformation system for differentiating C and C++, Procedia Computer Science 1 (1) (2010) 1845 – 1853, ICCS 2010. doi:DOI: 10.1016/j.procs.2010.04.206.

[15] Community portal for automatic differentiation, `http://www.autodiff.org`.

[16] D. Quinlan, ROSE Web Page, `http://rosecompiler.org`.

[17] M. Schordan, D. Quinlan, A source-to-source architecture for user-defined optimizations, in: JMLC'03: Joint Modular Languages Conference, Vol. 2789 of Lecture Notes in Computer Science, 2003, pp. 214–223.

[18] M. Tadjouddine, C. Faure, F. Eyssette, Sparse Jacobian computation in automatic differentiation by static program analysis, in: Lecture Notes in Computer Science, Vol. 1503, 1998, pp. 311–326.

[19] C. H. Bischof, A. Carle, P. Khademi, A. Mauer, ADIFOR 2.0: Automatic differentiation of Fortran 77 programs, IEEE Computational Science & Engineering 3 (3) (1996) 18–32.

[20] C. H. Bischof, L. Roh, A. Mauer, ADIC - An extensible automatic differentiation tool for ANSI-C, Software–Practice and Experience 27 (12) (1997) 1427–1456.

[21] Y. Kawajiri, L. Biegler, Large scale nonlinear optimization for asymmetric operation and design of Simulated Moving Beds, J. Chrom. A 1133 (2006) 226–240.

[22] M. Diehl, A. Walther, A test problem for periodic optimal control algorithms, Tech. Rep. MATH-WR-01-2006, TU Dresden (2006).

[23] Y. Kawajiri, L. Biegler, Large scale optimization strategies for zone configuration of simulated moving beds, Computers and Chemical Engineering 32 (2008) 135–144.

[24] C. H. Bischof, P. M. Khademi, A. Bouaricha, A. Carle, Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation, Optimization Methods and Software 7 (1997) 1–39.

[25] R. Giering, T. Kaminski, Automatic sparsity detection implemented as a source-to-source transformation, in: V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, J. Dongarra (Eds.), Computational Science – ICCS 2006, Vol. 3994 of Lecture Notes in Computer Science, Springer, 2006, pp. 591–598. doi:10.1007/11758549_81.

[26] A. Gebremedhin, A. Pothen, A. Walther, Exploiting sparsity in Jacobian computation via coloring and automatic differentiation: A case study in a simulated moving bed process, in: C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 327–338. doi:10.1007/978-3-540-68942-3_29.

[27] A. Walther, Computing sparse Hessians with automatic differentiation, ACM Trans. Math. Softw. 34 (1) (2008) 3:1–3:15.

[28] A. Gebremedhin, A. Pothen, A. Tarafdar, A. Walther, Efficient computation of sparse Hessians using coloring and automatic differentiation, INFORMS Journal on Computing 21 (2) (2009) 209–223.

[29] T. Coleman, B. Garbow, J. Moré, Software for estimating sparse Jacobian matrices, ACM Trans. Math. Softw. 10 (1984) 329–347.

[30] T. Coleman, B. Garbow, J. Moré, Software for estimating sparse Hessian matrices, ACM Trans. Math. Softw. 11 (1985) 363–377.

[31] M. Hasan, S. Hossain, T. Steihaug, DSJM: A Software Toolkit for Direct Determination of Sparse Jacobian Matrices, extended abstract at CSC09, SIAM Workshop on Combinatorial Scientific Computing (2009).

[32] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc Web page, http://www.mcs.anl.gov/petsc (2009).

[33] S. Balay, J. Brown, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 Revision 3.0.0, Argonne National Laboratory (2008).

[34] S. Balay, W. D. Gropp, L. C. McInnes, B. F. Smith, Efficient management of parallelism in object oriented numerical software libraries, in: E. Arge, A. M. Bruaset, H. P. Langtangen (Eds.), Modern Software Tools in Scientific Computing, Birkhäuser Press, 1997, pp. 163–202.

[35] P. Hovland, B. Norris, B. Smith, Making automatic differentiation truly automatic: Coupling PETSc with ADIC, in: P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, A. G. Hoekstra (Eds.), Computational Science – ICCS 2002, Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II, Vol. 2330 of Lecture Notes in Computer Science, 2002, pp. 1087–1096.