ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox

THOMAS F. COLEMAN and ARUN VERMA
Cornell University

ADMIT-1 enables the computation of *sparse* Jacobian and Hessian matrices, using automatic differentiation technology, from a MATLAB environment. Given a function to be differentiated, ADMIT-1 will exploit sparsity if present to yield sparse derivative matrices (in sparse MATLAB form). A generic automatic differentiation tool, subject to some functionality requirements, can be plugged into ADMIT-1; examples include ADOL-C (C/C++ target functions) and ADMAT (MATLAB target functions). ADMIT-1 also allows for the calculation of gradients and has several other related functions. This article provides an introduction to the design and usage of ADMIT-1.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Numerical algorithms*; G.1.5 [**Numerical Analysis**]: Roots of Nonlinear Equations—*Systems of equations*; G.1.6 [**Numerical Analysis**]: Optimization—*Unconstrained optimization*

General Terms: Algorithms

Additional Key Words and Phrases: Automatic differentiation, computational differentiation, efficient computation of gradient, graph coloring, Jacobians and Hessians, user interface

1. INTRODUCTION

The efficient numerical solution of nonlinear systems of algebraic equations $F(x) = 0$, $F(x) : \Re^n \rightarrow \Re^m$, usually requires the repeated calculation or estimation of the matrix of first derivatives, the Jacobian matrix, $J(x) \in \Re^{m \times n}$. In large-scale problems, the matrix J is often sparse, and it is important to exploit this fact in order to efficiently determine, or estimate, the matrix J at a given argument x.

The research was partially supported by the Cornell Theory Center which receives funding from Cornell University, New York State, the National Center for Research Resources at the National Institutes of Health, the Department of Defense Modernization Program, the United States Department of Agriculture, and members of the Corporate Partnership Program and the National Science Foundation under the grant DMS-9704509.
Authors' address: Computer Science Department and Cornell Theory Center, Cornell University, Ithaca, NY 14850; email: coleman@tc.cornell.edu.

Similarly, the efficient numerical solution of numerical optimization problems involving a scalar-valued function, $f(x) : \Re^n \rightarrow \Re$, may require repeated computation of the second derivative Hessian matrix $H(x) \in \Re^{n \times n}$. The symmetric matrix $H(x)$ is often sparse; it is important to exploit this sparsity in order to efficiently compute the matrix H at a given argument x.

In this article we present software to compute sparse Jacobian and Hessian matrices efficiently and painlessly using automatic differentiation-(henceforth referred to as AD) technology. ADMIT-1 is a MATLAB toolbox, which uses a generic AD plug-in tool (any AD tool can be used provided it satisfies the functionality criteria, which we describe in Section 4) to implement the sparse Jacobian and Hessian computing engines. The requirements from the users are minimal: the user is just required to supply the code for the function computation. For complete information on the ADMIT-1 toolbox, refer to the ADMIT-1 user manual [Coleman and Verma 1997]. There has been an earlier implementation of AD in MATLAB [Rich and Hill 1992] (thanks to the anonymous referee for pointing this out).

Automatic differentiation is a chain-rule-based technique for evaluating the derivatives analytically (and hence without any truncation errors) with respect to input variables of functions defined by a high-level language computer program [Berz et al. 1996; Griewank 1993; 1994; Griewank and Corliss 1991]. We present a basic review of automatic differentiation in Section 2.

Large-scale nonlinear problems often exhibit structure, e.g., partial separability, composition, discrete time optimal control forms, and inverse structure. The derivative matrices of these structured computations are typically dense; however, it is possible to define sparse *extended* derivative matrices [Coleman and Verma 1996a; 1996b] which can be computed using ADMIT-1. It is also possible to compute gradients (a special case of Jacobians) of structured computations by exposing the sparsity in an associated extended Jacobian matrix [Coleman and Jonsson 1999]. The software for structure computations is presented as a separate MATLAB toolbox, ADMIT-2 [Coleman and Verma 1999], an extension of the ADMIT-1 toolbox.

This article is outlined as follows. In Section 2, we give a brief background on AD followed by a review of the sparsity-exploiting techniques to compute the sparse Jacobian and Hessian matrices in Section 3. In Section 4 we present the software design of the ADMIT-1 tool. In Section 5, we present a detailed usage of the ADMIT-1 in a nonlinear equation solution using the Newton step. In Section 6 we present the algorithms and numerical results. In Section 7, we explore the different sparse derivative evaluation methods available in ADMIT-1 and provide examples on how to use them. In the Appendix we present brief description of the functionality of some main ADMIT-1 functions.

The ADMIT-1 software and related information can be accessed at `http://www.tc.cornell.edu/UserDoc/Software/Num/ad`.

2. AUTOMATIC DIFFERENTIATION BACKGROUND

Automatic differentiation is based on the fact that all computer programs, no matter how complicated, use a finite set of *elementary functions* as defined by the programming language. The function computed by the program is simply a composition of these elementary functions. The partial derivatives of the elementary functions are known, and the overall derivatives are computed using the chain rule; this process is known as automatic differentiation [Griewank 1993].

Abstractly, the program to evaluate the solution u (an m-vector) as a function of x (generally a n-vector) has the form

$$\begin{aligned} x &\equiv (x_1, x_2, \ldots, x_n) \\ &\downarrow \\ z &\equiv (z_1, z_2, \ldots, z_p), p \gg m + n \\ &\downarrow \\ y &\equiv (y_1, y_2, \ldots, y_m) \end{aligned}$$

where the intermediate variables z are related through a series of these elementary functions which may be unary,

$$z_k = f^k_{\text{elem}}(z_i), \quad i < k,$$

consisting of operations such as $(-, \text{pow}(\cdot), \sin(\cdot), \ldots)$ or binary,

$$z_k = f^k_{\text{elem}}(z_i, z_j), \quad i < k, \quad j < k.$$

such as $(+, /, \ldots)$.

There are a number of cases when the elementary function is not differentiable (e.g., $f^k_{\text{elem}}(z_i) = \text{abs}(z_i)$ or $f^k_{\text{elem}}(z_i, z_j) = \max(z_i, z_j)$). Sophisticated heuristic techniques are developed to treat these cases. For more details consult Griewank [1993].

Automatic Differentiation has two basic modes of operations, the forward mode and the reverse mode. In the forward mode the derivatives are propagated throughout the computation using the chain rule, e.g., for the elementary step $z_k = f^k_{\text{elem}}(z_i, z_j)$ the intermediate derivative dz_k/dx can be propagated in the forward mode as

$$\frac{dz_k}{dx} = \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{dz_i}{dx} + \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{dz_j}{dx}.$$

This chain-rule-based computation is done for all the intermediate variables z and for the output variables u, finally yielding the derivative du/dx.

The reverse mode computes the derivatives du/dz_k for all intermediate variables backward (i.e., in the reverse order) through the computation. For example, for the elementary step $z_k = f^k_{elem}(z_i, z_j)$, the derivatives are propagated as

$$\frac{du}{dz_i} += \frac{\partial f^k_{\text{elem}}}{\partial z_i}\frac{du}{dz_k} \quad \text{and} \quad \frac{du}{dz_j} += \frac{\partial f^k_{\text{elem}}}{\partial z_j}\frac{du}{dz_k}.$$

At the end of computation of the reverse mode the derivative du/dx will be obtained. The derivatives in the adjoint mode are propagated in an incremental form in the adjoint mode because the arguments of the elementary function may appear again in the forward evaluation process; all the derivatives are initialized to zero.

The forward and reverse modes can be used to compute the direct and the adjoint products Jv and J^Tv given a vector v, where J is the Jacobian of a nonlinear mapping [Griewank 1993]. Both these computations require time proportional to one function evaluation, with the adjoint product being approximately twice as costly as the direct mode. The Hessian-vector product Hv can also be computed via AD in time proportional to one function evaluation.

3. COMPUTATION OF SPARSE JACOBIAN AND HESSIAN MATRICES

In this section we review the techniques for computing sparse Jacobian and Hessian matrices. For details on this subject refer to Coleman and Cai [1986], Coleman and Verma [1998b], and Coleman and Moré [1984a; 1984b].

Sparse finite-differencing techniques were first introduced by Curtis et al. [1974], Coleman and Moré [1984a; 1984b], and Coleman and Cai [1986], and Newsam and Ramsdell [1983] further developed these ideas using graph-theoretic interpretations. Recently, related methods were developed to be used in conjunction with AD tools instead of finite differencing [Averick et al. 1994; Bischof et al. 1997; Coleman and Verma 1998a].

3.1 Computation of a Sparse Jacobian

One way to approach the problem of estimating a sparse Jacobian matrix of a mapping $F : \Re^n \to \Re^m$ is in the following terms: given a sparse $m \times n$ matrix J, obtain vectors $d_1, d_2, \ldots, d_p$ such that the products $Jd_1, Jd_2, \ldots, Jd_p$ determine J uniquely. For example, if J is diagonal, then $d_1 = e$ (a vector of all ones) suffices, since Je determines all nonzeros of J uniquely. If J is tridiagonal, then only three products are required, Jd_1, Jd_2, and Jd_3 where $d_1 = e_1 + e_4 + e_7 + \ldots$, $d_2 = e_2 + e_5 + e_8 + \ldots$, $d_3 = e_3 + e_6 + e_9 + \ldots$. The matrix J can then be reconstructed because each nonzero entry of J appears in one of Jd_1, Jd_2 or Jd_3. This approach is called the one-sided column approach for computing a sparse Jacobian [Coleman and Moré 1984b; Coleman et al. 1984; Curtis et al.

1974]. The alternative row approach can be phrased as "obtain vectors d_1, d_2, . . . , d_p such that the products $J^T d_1$, $J^T d_2$, . . . , $J^T d_p$ determine J uniquely." The vectors d_i are determined by the nonzero structure of J. This method cannot be implemented using finite differences based on F; however, AD can be used in the reverse mode to compute products $J^T d$.

The new bicoloring approach [Coleman and Verma 1998a], which combines the row and column views, is an efficient approach for minimizing the cost of computing a sparse Jacobian matrix of a nonlinear map, employing AD. The authors show how to define "thin" matrices V and W such that the nonzero elements of J can easily be extracted from the calculated pair $(W^T J, JV)$. The pair $(W^T J, JV)$ can be directly computed using AD given an arbitrary n-by-t_V matrix V and and an arbitrary m-by-t_W matrix, employing the forward mode for computing JV and the reverse mode for computing $W^T J$. A similar approach outlining the computation of the sparse Jacobian using rows and columns was given by Hossain and Steihaug [1995].

The motivation for taking this two-sided view comes from the following observations. The one-sided column solution based on a column partition defines a matrix V such that J can be determined from the product JV. However, matrix V is not guaranteed to be thin, even if J is very sparse: consider a sparse matrix J with a single dense row. Alternatively, a solution based on partitioning of rows can be employed to define a matrix W such that J can be determined from $W^T J$. Again, it is easy to construct examples where defining a thin W is not possible: e.g., consider the case where J has a single dense column.

Bicoloring circumvents this problem and is never worse than one-sided coloring. Here is a simple example which demonstrates the advantage of bicoloring. Consider the following n-by-n Jacobian, symmetric in structure but not in value:

$$J = \begin{pmatrix} \square & \triangle & \triangle & \triangle & \triangle \\ \square & \diamond & & & \\ \square & & \diamond & & \\ \square & & & \diamond & \\ \square & & & & \diamond \end{pmatrix} \tag{1}$$

It is clear that a partition of columns consistent with the direct determination of J requires n groups. This is because a "consistent column partition" requires that each group contain columns that are structurally orthogonal, and the presence of a dense row implies each group consists of exactly one column. Therefore, if matrix V corresponds to a "consistent column partition" then V has n columns and the work to evaluate JV by the forward mode of AD is proportional to $n \cdot \omega(F)$. By a similar argument, and the fact that a column of J is dense, a "consistent row partition" requires n groups.

Table I. Totals for LP Collection (`http://www.netlib.org/lp/data/`)

Bicoloring		One-Sided Coloring	
Direct	Substitution	Column	Row
337	270	1753	452

Table II. Totals for Harwell-Boeing Collection (ftp from `orion.cerfacs.fr`)

Bicoloring		One-Sided Coloring	
Direct	Substitution	Column	Row
320	244	732	738

Therefore, if matrix W corresponds to a "consistent row partition" then W has n rows, and the work to evaluate W^TJ by the reverse mode of AD is proportional to $n \cdot \omega(F)$. In this example the use of a bicoloring dramatically decreases the amount of work required to determine J. Specifically, the total amount of work required is proportional to $3 \cdot \omega(F)$. To see this define $V = (e_1, e_2 + e_3 + e_4 + e_5)$; $W = (e_1)$, where we follow the usual convention of representing the ith column of the identity matrix with e_i. Clearly elements $\square$, $\diamond$ are directly determined from the product JV; elements $\triangle$ are directly determined from the product W^TJ.

A graph-theoretic interpretation of the determination problem can be constructed based directly on Jacobian structure. The associated graph-coloring problems are known to be NP-complete [Coleman and Verma 1998a; Garey and Johnson 1979]; therefore, heuristic schemes are considered to construct the "bipartition." For more insight into the problem involved and algorithmic details refer to Section 6.

Below are performance results obtained for bicoloring summarized from Coleman and Verma [1998a]: Table I shows the summary of the performance of bicoloring on a linear programming testbed of matrices; Table II shows the performance on the Harwell-Boeing collection. The numbers in the tables denote the total number of Jacobian matrix products (forward Jd or adjoints J^Td) needed to compute the sparse Jacobian matrices in the collection.

Also, similar to the results reported in Averick et al. [1994] for forward-mode direct determination, the Jacobian matrices determined by our bicoloring/AD approach are significantly and uniformly more accurate than the finite-difference approximations (usually around 6 digits more accuracy than FD). This is true for both direct determination and the substitution approach. Second, the direct approach is uniformly more accurate than the substitution method. The Jacobian matrices determined via substitution are sufficiently accurate for most purposes, achieving at least 10 digits of accuracy and usually more. For comparison on accuracy of these methods we refer the reader to Coleman and Verma [1998a].

In Section 6, we present an overview of implementation of the one-sided column and row methods and the bicoloring method in ADMIT-1 software.

3.2 Computation of a Sparse Hessian

In this section we review the techniques to compute sparse Hessian matrices. It is well known that the product $\nabla^2 f(x) \cdot d$ can be computed using AD, or approximated by finite differencing. When the nonzero structure of $\nabla^2 f(x)$ is known, then usually a few well-chosen directions d_1, d_2, . . . , d_p are needed to compute all the nonzeros of $\nabla^2 f(x)$ using the products $\nabla^2 f(x)d_1$, $\nabla^2 f(x)d_2$, . . . , $\nabla^2 f(x)d_p$.

The algorithms that we have implemented are based on the work of Powell and Thoint [1979], Coleman and Cai [1986], Coleman and Moré [1984a], and Coleman et al. [1984]. These authors consider direct and indirect (substitution) methods; indirect methods usually require fewer function (or gradient) evaluations, while direct methods produce more accurate approximations to the Hessian matrix H. For a complete review on this subject, refer to Coleman and Cai [1986].

Let G represent the adjacency graph of H. In summary, there are basically three different ways to compute a sparse Hessian:

(1) *Ignoring the symmetry:* This is exactly like the single-sided Jacobian problem: symmetry is ignored, i.e., $H_{i,j}$ and $H_{j,i}$ are computed independently. Since, the intersection graph of H is given by the adjacency graph of matrix H^2, the minimum number of groups needed to compute the Hessian via this method is denoted by $\chi(G^2)$. The intersection graphs and their construction are explained in detail in Section 6.

(2) *Direct—exploiting symmetry:* This is the path-coloring method as described in Coleman and Moré [1984a]. The minimum number of function evaluations are given by the path-coloring chromatic number which is denoted by $\chi_\pi(G)$.

(3) *Substitution—exploiting symmetry:* This is the cyclic-coloring method as described in Coleman and Cai [1986]. The complexity in this case is given b ythe cyclic-coloring chromatic number which is denoted by $\chi_0(G)$.

Since every cyclic coloring of G is a coloring of G, and every coloring of G^2 is a path coloring of G, and a path coloring a cyclic coloring, we get the following string of inequalities:

$$\chi(G) \leq \chi_0(G) \leq \chi_\pi(G) \leq \chi(G^2)$$

ADMIT-1 software provides methods for computing the sparse Hessian matrix using any of the three methods. We present an example in Section 6 which illustrates the selection of any of above three methods for computing a Hessian matrix.

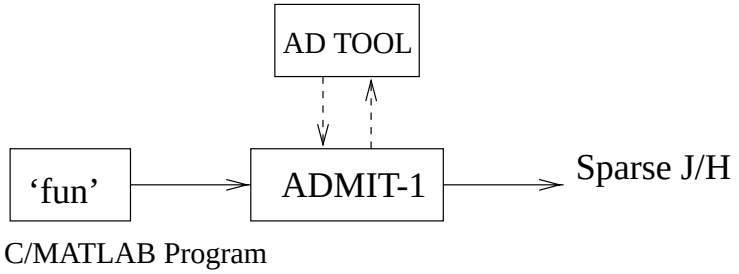

Fig. 1. Design of ADMIT-1 toolbox.

4. SOFTWARE DESIGN OF ADMIT-1

The high-level design of the ADMIT-1 toolbox is shown in Figure 1. ADMIT-1 takes an AD tool as a plug in (shown in the top box) and takes as input the user-defined nonlinear function ("fun") and outputs a sparse Jacobian or Hessian matrix as required by the user. A generic AD tool, with functionality described in Section 4.1, is required.

The core of the ADMIT toolbox is formed by two routines, evalJ and evalH, with usage described in Appendix A. ADMIT-1 uses the sparse techniques for computation of sparse Jacobian and Hessians as outlined in Section 3 (and other derivative information like gradient and Jacobian matrix products etc.). Refer to the Appendix to learn about additional functionality of ADMIT-1.

4.1 Expected Design of the Underlying AD Tool

The underlying AD plug-in tool is expected to have both reverse and forward modes of automatic differentiation. If the AD tool has only the forward mode then it can be used with ADMIT-1, but cannot take advantage of the bicoloring technique to evaluate the sparse Jacobian matrix. In particular, the following five capabilities from the AD tool are recommended in order to qualify as a plug-in tool for ADMIT-1 (first three for the Jacobian/gradient evaluations and the last two for the Hessian evaluation). ADMIT-1 requires the source file (e.g., C or MATLAB, depending on target functionality of the plug-in AD tool) for the input function. The design of the user function is outlined in Section 5.

The five functionality features expected from the AD tool are listed below. The function $f : \Re^n \to \Re$ is a scalar-valued function, and $F : \Re^n \to \Re^m$ is a vector-valued function.

(1) *Jacobian-Matrix (forward) product:* $(F, x, V) \to J(x)V$.

(2) *Matrix-Jacobian (reverse) product:* $(F, x, W) \to J(x)^T W$.

(3) *Jacobian Sparsity Pattern:* $F \to SPJ$.

(4) *Hessian-Matrix product:* $(f, x, V) \to H(x)V$.

(5) *Hessian Sparsity Pattern:* $f \to SPH$.

SPJ and *SPH* stand for the sparsity patterns of the Jacobian and Hessian matrix respectively. ADMIT-1 requires that the AD plug-in tool compute these sparsity patterns automatically.

Note that gradient computation is a special case of these requirements, since computing the gradient is equivalent to a reverse product with $W = 1$, a scalar; the reverse product is $\nabla f(x) = J^T$.

The packages ADOL-C [Griewank et al. 1996] and ADMAT [Coleman and Verma 1998a] satisfy the requirements listed above.

The sparse Jacobian and Hessian matrices can be computed using a method of your choice, e.g., bicoloring. ADMIT-1 provides ADOL-C drivers for the above functions, described in Appendix B. An AD tool which implements only the forward mode can also be used as an ADMIT-1 plug-in, albeit with restricted features. For example, the bicoloring technique is replaced with the one-sided column method. If you need to compute only the first-order derivatives, then an AD tool which has the first three features can be used as an ADMIT-1 plug-in.

5. EXAMPLE OF ADMIT-1 USAGE FOR SOLVING NONLINEAR EQUATIONS

In this section we illustrate a local nonlinear equation solver, based on a sequence of Newton steps, using ADMIT-1.

5.1 User Function Design

First, we describe the design of the functions that can be used with ADMIT-1. Here we present the expected designs of C/C++ target functions (with ADOL-C as the plug-in AD tool) and MATLAB target functions (with ADMAT as the plug-in AD tool).

If ADOL-C is the plug-in AD tool, the design of the target C/C++ function is as follows. The function must be named `getfun`.

```
void getfun(float* x,int n,float* y,int m, float *Extra, int
*numrows, int *numcols)
{
/* Compute y = F(x) here */
}
```

The input argument `x` is a vector of dimension `n`; `y` is the output vector of dimension `m`. `Extra` is a one-dimensional array corresponding to a two-dimensional (full) matrix stacked column-by-column. The matrix represented by `Extra` is of size `numrows`-by-`numcols`.

If ADMAT is the plug-in AD tool, the design of the target MATLAB function is as follows:

```
function y = getfun(x,Extra)
% Compute y =F(x) here
end
```

Here is a simple example illustrating how to use ADMIT-1 to calculate the Jacobian of the function $y = F(x)$, $F : \Re^n \rightarrow \Re^n$ where

$$y(1) = 2x(1)^2 + \sum_{1}^{n} x(i)^2,$$

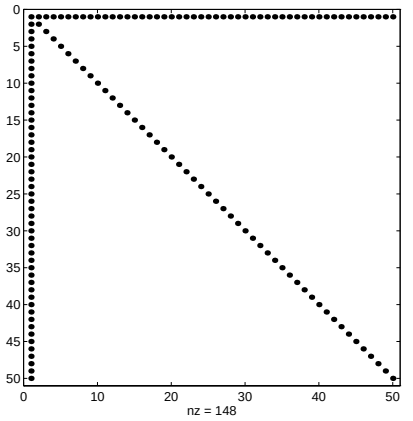

Fig. 2. The sparsity structure of Jacobian J.

$$y(i) = x(i)^2 + x(1)^2, \quad i = 2 : n.$$

The Jacobian of function F has an arrowhead sparsity structure, as shown in Figure 2 for $n = 50$.

When using ADOL-C as the plug-in AD tool, ADMIT-1 requires a C program (`examplefun.c`) to evaluate F:

```
void getfun(float* x,int n,float* y,int m, float *Extra, int
*numrows, int *numcols)
{
  int j;
  /* Nonzero Diagonal */
  for(j=0;j;j++)
    y[j]=x[j]*x[j];
  for(j=0;j;j++)
  {
    /* Dense first row */
    y[0]=y[0]+x[j]*x[j];
    /* Dense first column */
    y[j]=y[j]+x[0]*x[0];
  }
}
```

In order to evaluate the function F and the Jacobian J at $x' = (1, 1, \ldots, 1)$ for $n = 5$ execute in MATLAB:

```
>> x=ones(5,1); n = 5;
>> fun='examplefun'; JPI = getJPI(fun,n);
>> [f,J]=evalJ(fun,x,[],[],JPI);
>> f
f =
     7
     2
     2
     2
     2
```

The function call "getJPI" extracts sparsity/coloring information. As illustrated in the Newton iteration example in Section 5.2, only one execution of "getJPI" is required for a given target function.

To use ADMAT as the plug-in AD tool instead of ADOL-C, the same MATLAB script for evaluating the Jacobian can be used with the MATLAB version of the function:

```
function f= examplefun(x,m,Extra)
       y=x.*x;
       y(1)=y(1)+x'*x;
       y=y+x(1)*x(1);
```

5.2 Newton Process for Nonlinear Equations F(x)=0

Suppose that the user has a MATLAB routine to compute the nonlinear function $F(x)$ and needs to solve $F(x) = 0$ for the vector x. A typical method is to employ the Newton iteration method, and we illustrate this method using ADMIT-1 via an example. The example target function is the "Broyden" nonlinear function. Here is the shell (MATLAB) program:

```
>> fun = 'broyden';
>> itbnd=100;
>> tol= 1e-6;
>> xstart=[zeros(50,1);0.2*ones(50,1)];
>>
>> %get the Coloring Info Once and for all
>> JPI= getJPI(fun,100);
>>
>> [x,it,norm] = newton(fun,xstart,tol,itbnd,JPI);
>> cleanup
>> exit
```

The Broyden nonlinear function is listed here:

```
function fvec= broyden(x,Extra);
% Evaluate the Broyden nonlinear equations test function.
       n = length(x); fvec=zeros(n,1);
       i=2:(n-1);
       fvec(i)= (3-2*x(i)).*x(i)-x(i-1)-2*x(i+1)+ones(n-2,1);
       fvec(n)= (3-2*x(n)).*x(n)-x(n-1)+1;
       fvec(1)= (3-2*x(1)).*x(1)-2*x(2)+1;
```

Finally we list our M-file containing the Newton procedure.

```
function [x,it,nf]= Newton(fun, xstart, tol, itbnd, JPI)
% Initializations
n=length(xstart);
if (nargin < 3) tol=1e-5; end
if (nargin < 4) itbnd=60; end
n=length(xstart); x=xstart;

% First Evaluation
[f,J]=evalJ(fun,x,[],[],JPI);
it=0;
```

Table III. Various Methods for Computing Sparse Jacobian Matrices

Method	Chromatic Number Notation
One-sided column method	$\chi_c(J)$
One-sided row method	$\chi_r(J)$
Finite-differencing method	$\chi_f(J)$
Direct bicoloring method	$\chi_d(J)$
Substitution bicoloring method	$\chi_s(J)$

```
% The Newton Iteration
while ((norm(f) > tol) & (it < itbnd))
       delta= -J \f;
       x=x+delta;
       [f,J]=evalJ(fun,x,[],[],JPI);
       it=it+1;
end
nf=norm(f);
```

The function `Newton` uses the ADMIT-1 driver function `evalJ` to compute the Jacobian matrix and use it in a subsequent Newton step computation. ADMIT-1 functions can be plugged into optimization algorithms to provide an efficient and accurate solution to nonlinear problems.

6. ALGORITHMS

In Section 2 we reviewed the techniques for computing sparse Jacobian and Hessians used in the ADMIT-1 software. In this section, we present the algorithms involved in implementing the graph-theoretic techniques.

6.1 Computing Sparse Jacobians

There are basically five different options to compute a sparse Jacobian matrix using ADMIT-1. These different methods correspond each to a different partition which can be computed by solving a graph-coloring problem on appropriately defined graphs [Coleman and Moré 1984b]. The chromatic number of a graph is defined as the least number of colors required to color the graph, or in other words, the least number of groups required to compute the Jacobian matrix. The method and corresponding chromatic numbers are illustrated in Table III.

The algorithm involved for the finite-differencing method is the same as the one-sided column AD method except that the former uses finite differences to approximate the product Jd. The various chromatic numbers satisfy the inequality:

$$\chi_s(J) \leq \chi_d(J) \leq \min(\chi_c(J), \chi_r(J)) \tag{2}$$

Inequality (2) holds, since bicoloring subsumes both one-sided coloring techniques; for more details, refer to Coleman and Moré [1984b].

6.1.1 *The One-Sided Algorithms.* We first review the algorithms for the one-sided methods. The one-sided column method involves coloring the

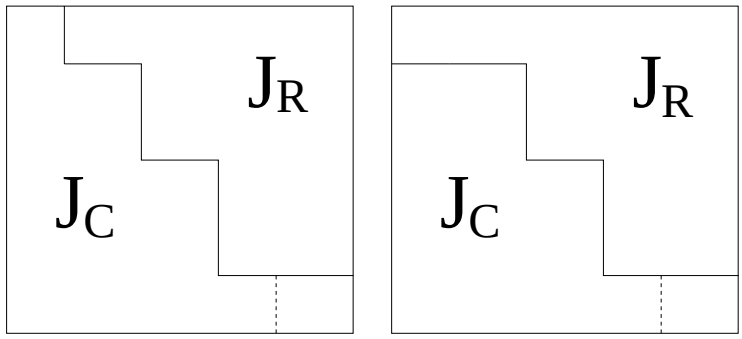

Fig. 3. Possible partitions of the matrix $\tilde{J} = P \cdot J \cdot Q$.

column intersection graph of the Jacobian sparsity structure. For a detailed treatment on this subject, please refer to Coleman and Moré [1984b]. The method implemented in ADMIT-1 is outlined in the following pseudo-MATLAB code:

```
function group = color(J);
     [m,n] = size(J);
     ng=0;
     while there are ungrouped columns
          find an ungrouped column c;
          for i= 1: ng
            if c doesn't intersect with any column in group i
              assign it group i: group(c)=i;
            end
          end

          if c is unassigned, assign it a new group:
            ng=ng+1; group(c)=ng+1;
          end
     end
end
```

In the above code, J denotes the sparsity structure of the Jacobian matrix. Two columns are said to "intersect" if they both have a nonzero in the same row position. The main step of the algorithm consists of assigning each vertex in turn the lowest numbered color not yet used by the neighbors. The order in which the candidate columns are searched for is unspecified in the algorithm given above. Ordering based on graph-coloring heuristics has proven to be effective [Coleman et al. 1984]. One such ordering, smallest-degree ordering, is the default ordering used in ADMIT-1.

The one-sided column method is just the transpose of column method: the same coloring algorithm is used on the sparsity pattern of J^T.

6.1.2 *The Bicoloring Algorithms*. The problems of finding the best "bi-partition" for both direct and substitution determination can be approached in the following way. First, permute and partition the structure of J: $\tilde{J} = P \cdot J \cdot Q = [J_C | J_R]$, as indicated in Figure 3. The construction of this

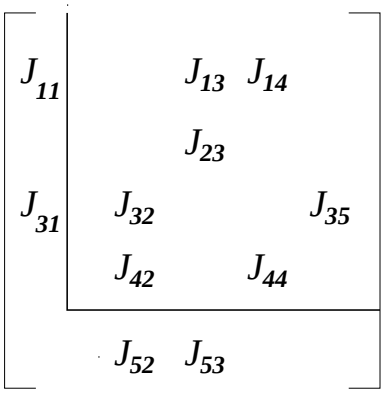

Fig. 4. Example partition.

partition is crucial; however, we postpone that discussion until after we illustrate its utility. Assume $P = Q = I$ and $J = [J_C | J_R]$.

Second, define appropriate intersection graphs $\mathcal{G}_C^I$, $\mathcal{G}_R^I$ based on the partition $[J_C | J_R]$; a coloring of $\mathcal{G}_C^I$ yields a partition of a subset of the columns, G_C, which defines matrix V. Matrix W is defined by a partition of a subset of rows, G_R, which is given by a coloring of $\mathcal{G}_R^I$. The difference between the direct and substitution cases is in how the intersection graphs, $\mathcal{G}_C^I$, $\mathcal{G}_R^I$, are defined, and how the nonzeroes of J are extracted from the respective pair, $(W^T J, JV)$.

For this discussion, we omit the details on how the intersection graphs $\mathcal{G}_C^I$, $\mathcal{G}_R^I$ are defined, for both the direct and substitution bicoloring. For the algorithmic details, refer to Coleman and Verma [1998a]. Once the intersection graphs are colored, the boolean matrices V and W can be formed in the usual way: each column corresponds to a group (or color), and unit entries indicate column (or row) membership in that group:

Example. Consider the example Jacobian matrix structure shown in Figure 4 with the partition (J_C, J_R) shown.

The matrices V and W for this problem turn out to be

$$V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad JV = \begin{pmatrix} J_{11} & 0 & \times \\ 0 & 0 & \times \\ J_{31} & \times & \times \\ 0 & \times & 0 \\ 0 & J_{52} & J_{53} \end{pmatrix}$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad W^T J = \begin{pmatrix} \times & J_{32} & J_{13} & J_{14} & J_{35} \\ \times & J_{42} & J_{23} & J_{44} & 0 \end{pmatrix}.$$

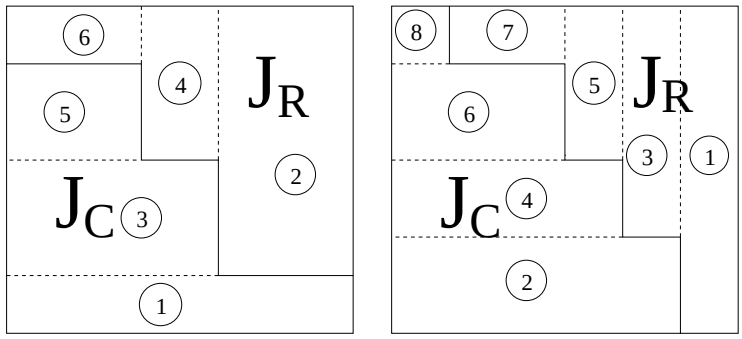

Fig. 5. Substitution orderings.

Clearly, all nonzero entries of J can be identified in either JV or W^TJ.

Determination by Substitution. The basic advantage of determination by substitution in conjunction with partition $J = [J_C|J_R]$ is that sparser intersection graphs $\mathcal{G}_C^I$, $\mathcal{G}_R^I$ can be used. Sparser intersection graphs mean thinner matrices V, W which, in turn, result in reduced cost.

All the elements of J can be determined from (W^TJ, JV) by a substitution process. This is evident from the illustrations in Figure 5.

Figure 5 illustrates two of four possible nontrivial types of partitions. The nonzero elements in the section labeled "1" can be solved for directly—by the construction process there will be no conflict. Nonzero elements in "2" can either be determined directly, or will depend on elements in section "1." But elements in section "1" are already determined (directly), so, by substitution, elements in "2" can be determined after "1." Elements in section "3" can then be determined, depending only on elements in "1" and "2," and so on, until the entire matrix is resolved.

Example. Consider again the example Jacobian matrix structure shown in Figure 4.

The coloring of $\mathcal{G}_C$ and $\mathcal{G}_R$ leads to the following matrices V, W and the resulting computation of JV, W^TJ:

$$V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \quad JV = \begin{pmatrix} J_{11} + J_{13} & 0 \\ \times & 0 \\ J_{31} & \times \\ 0 & \times \\ J_{53} & J_{52} \end{pmatrix}$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad W^TJ = \begin{pmatrix} \times & J_{32} & J_{13} & J_{14} & J_{35} \\ \times & J_{42} & J_{23} & J_{44} & 0 \end{pmatrix}$$

It is now easy to verify that all nonzeroes of J can be determined via substitution.

How to Partition J. We now consider the problem of obtaining a useful partition $[J_C | J_R]$, and corresponding permutation matrices P, Q, as illustrated in Figure 3.

Algorithm **MNCO** builds partition J_C from bottom up, and partition J_R from right to left. At the kth major iteration either a new row is added to J_C, or a new column is added to J_R; the choice depends on considering a lower bound effect:

$$\rho(J_R^T) + \max(\rho(J_C), nnz(r)) < (\rho(J_C) + \max(\rho(J_R^T), nnz(c))), \qquad \text{(LB)}$$

where $\rho(A)$ is the maximum number of nonzeroes in any row of matrix A; r is a row under consideration to be added to J_C; and c is a column under consideration to be added to J_R. Hence, the number of colors needed to color $\mathcal{G}_C^I$ is bounded below by $\rho(J_C)$; the number of colors needed to color $\mathcal{G}_R^I$ is bounded below by $\rho(J_R^T)$.

In algorithm **MNCO**, matrix $M = J(R, C)$ is the submatrix of J defined by row indices R and column indices C: M consists of rows and columns of J not yet assigned to either J_C or J_R.

Minimum Nonzero Count Ordering (MNCO).

(1) Initialize $R = (1 : m)$, $C = (1 : n)$, $M = J(R, C)$
(2) Find $r \in R$ with fewest nonzeroes in M
(3) Find $c \in C$ with fewest nonzeroes in M
(4) Repeat Until $M = \varnothing$
  if $\rho(J_R^T) + \max(\rho(J_C), nnz(r)) < (\rho(J_C) + \max(\rho(J_R^T), nnz(c)))$ (LB)
    $J_C = J_C \cup (r \cap C)$
    $R = R - \{r\}$
  else
    $J_R = J_R \cup (c \cap R)$
    $C = C - \{c\}$
  end if
  $M = J(R, C)$.
end repeat

Note that J_R, J_C, upon completion, have been defined; the requisite permutation matrices are implicitly defined by the ordering chosen in **MNCO**.

6.1.3 *Numerical Results.* We give some results here to illustrate the effectiveness of the bicoloring technique. The test function $F \in \Re^{n \times n}$ we use is a sample nonlinear function which has a sparse Jacobian matrix having the structure shown in Figure 2. Additional details are provided in Coleman and Verma [1998a]. Our results, shown in Figure 6, suggest the following order of execution time requirement by different techniques for the given test function:

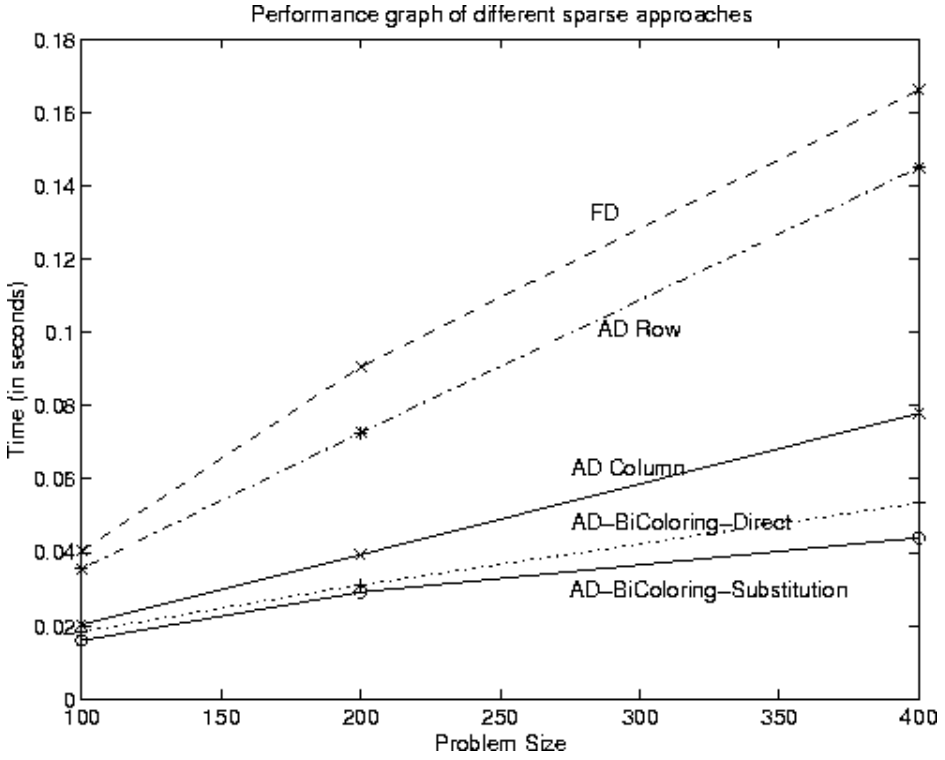

Fig. 6. A comparison of different sparse techniques.

$$FD > AD(row) > AD(column) > AD(bi\text{-}coloring, direct) > AD(bi\text{-}coloring, substitution).$$

In general, the above order will vary a little depending on the problem being solved, e.g., for the special case of computing gradient, typically the $AD(row)$ method will be quicker than the $AD(column)$. However, the above order for the given test function is typical for general nonlinear functions, e.g., for the problems with results summarized in Tables I and II.

FD stands for the finite-differencing method. $AD(row)$, $AD(column)$ are the one-sided methods. Note that FD requires more time than $AD(column)$ even though the same coloring is used for both. This is because the work estimate $t_V \cdot \omega(F)$ is actually an upper bound on the work required by the forward mode where t_V is the number of columns of V. In contrast, $t_V \cdot \omega(F)$ is tight for finite differencing, since the subroutine to evaluate F is actually called (independently) t_V times.

Another interesting observation is that the reverse-mode calculation, $AD(row)$, is about twice as expensive as the forward calculation $AD(column)$. This is noteworthy because, in this example, based on the structure in Figure 1, the column dimensions of V and W are equal. It may

be pragmatic to estimate "weights" w_1, w_2, with respect to a given AD tool, reflecting the relative costs of forward and reverse modes. It is very easy to introduce weights into algorithm **MNCO** to heuristically solve a "weighted" problem. The heuristic **MNCO** can be changed to address this problem by simply changing the conditional (LB) to

$$\text{if } w_1 \cdot \rho(J_R^T) + w_2 \cdot \max(\rho(J_C), nnz(r))$$

$$< w_1 \cdot \rho(J_C) + w_2 \cdot \max(\rho(J_R^T), nnz(c)).$$

6.2 Algorithms for Computing Sparse Hessians

The algorithms we have implemented for this step are based on the work of Powell and Thoint [1979] and Coleman et al. [1985].

(1) *Ignoring the symmetry:* Given the sparsity pattern of Hessian, SPH, subroutine `ignhess` (called by getHPI) determines a permutation p and a partition of the columns of H, consistent with the determination of all nonzeros H directly and independently. This routine is the same as the one used for the one-sided column method for the Jacobians.

(2) *Direct—exploiting symmetry:* Given the sparsity pattern of Hessian, SPH, subroutine `dirhess` (called by getHPI) determines a permutation p and a partition of the columns of H, consistent with the determination of all nonzeros H directly and exploiting the symmetry of H.

This method implements path coloring [Coleman et al. 1985]:

Path-coloring algorithm
Let $G = (V, E)$ be the adjacency graph.
for $k = 1, 2, \ldots$
(a) Let U_k be the set uncolored vertices. If U_k is empty, STOP.
(b) Sort the vertices in $G(U_k)$, in decreasing order of degree.
(c) Build a vertex set W_k, by examining the vertices in U_k in the order determined in step 2, and adding a vertex v to W_k, if there is not a path between v and any vertex in W_k of length ≤ 2.
(d) for each v in W_k, assign $color(v) = k$.
endfor

The array *color* determines the grouping of columns.

(3) *Substitution—exploiting symmetry:* Given the sparsity pattern of Hessian, SPH, subroutine `subhess` (called by getHPI) determines a permutation p and a partition of the columns of H, consistent with the determination of all nonzeros H by substitution.

This method requires cyclic coloring of the adjacency graph of the Hessian matrix. The algorithm involved can be found in detail in Coleman and Cai [1986]. In summary, the algorithm involves finding a

permutation π, such that the columns of L_π (the lower triangular part of $H(\pi, \pi)$) in the same group do not intersect in the same row position. So the algorithm involves two main steps:

(a) Find a permutation (using a heuristic scheme such as the symmetric minimum degree method) π, such that the column intersection graph of $L\pi$ is as sparse as possible.

(b) Color the intersection graph $G(L_\pi)$ to yield the column grouping g.

7. CHOOSING DIFFERENT COLORING METHODS

ADMIT-1 allows for usage of different coloring `method` options, via the two functions, `getJPI` and `getHPI`. In the following illustration, we demonstrate how to use different methods.

```
>> m=100; n=100;
>> JPId = getJPI(fun,m,n);  ← JPI for direct bicoloring
>> (default) method
>> JPIs = getJPI(fun,m,n,[],'s');  ← JPI for substitution
>> bicoloring method
>> JPIc = getJPI(fun,m,n,[],'c');  ← JPI for column coloring
>> method
```

In the above illustration the sparsity pattern of the Jacobian is computed three times. This is costly and it can be avoided:

```
>>....
>>
>> [JPId,SPJ] = getJPI(fun,m,n);  ← JPI for direct bicoloring
>> (default) method
>> JPIs = getJPI([],m,n,[],'s',SPJ);  ← JPI for substitution
>> bicoloring method
>> JPIc = getJPI([],m,n,[],'c',SPJ);  ← JPI for column
>> coloring method
```

Similarly for Hessians:

```
>> n=100;
>> [HPIi,SPH] = getHPI(fun,n);  ← HPI for ignore symmetry
>> (default) method
>> HPId = getHPI([],n,[],'d-a',SPH);  ← HPI for direct
>> symmetry exploiting method
>> HPIs = getHPI([],n,[],'s-a',SPH);  ← HPI for substitution
>> symmetry exploiting method
```

8. CONCLUDING REMARKS

The ADMIT-1 toolbox extends the MATLAB environment to provide a powerful computing environment for large-scale optimization and sensitivity analysis. The capability of doing automatic differentiation within the MATLAB opens up a wide range of applications which can easily use the AD technology.

The use of the ADMIT-1 tool with ADMAT as the plug-in tool is particularly interesting. Since ADMAT is written in MATLAB, it can be

used just like any MATLAB toolbox without the need of external compilation steps (unlike the ADOL-C plug-in). Also, ADMAT can be readily applied to any of the thousands of different functions present in the MATLAB toolbox. With ADMAT, it is now possible to differentiate through a variety of MATLAB toolboxes, thus enabling automatic differentiation of complicated MATLAB applications. (However, ADMAT cannot differentiate through MEX files, since it needs the full MATLAB source code.)

APPENDIX

A. THE ADMIT-1 FUNCTIONALITY

In this section, we illustrate the high-level functionality of ADMIT-1. First, we describe the main functions `evalJ` and `evalH` both of which employ the algorithms discussed in previous sections.

evalJ

Purpose.

Compute the value of a differentiable vector mapping f and its Jacobian J. Function `evalJ` is designed for the case where J is a sparse matrix.

Synopsis.

```
f=evalJ(fun,x)
f=evalJ(fun,x,Extra)
f=evalJ(fun,x,Extra,m)
[f,J]=evalJ(fun,x,Extra,m,JPI)
[f,J]=evalJ(fun,x,Extra,m,JPI,verb)
[f,J]=evalJ(fun,x,Extra,m,JPI,verb,fdstep)
```

Description.

`f=evalJ(fun,x,Extra,m)` Evaluate the function at the input argument `x`. The function is assumed to be a square mapping with dimension defined by the length of `x`. The first input argument, `fun`, is an integer handle identifying the target function. You can provide a *full* matrix, `Extra`, to be used by your target function. `Extra` cannot be a MATLAB *sparse* matrix. Scalar `m` is the row dimension of the vector mapping, i.e., $f : \Re^n \rightarrow \Re^m$.

`[f,J]=evalJ(fun,x,Extra,m,JPI)` Evaluate the sparse Jacobian `J` at the point `x`. `JPI` encodes the "coloring" information about the sparse matrix J. (See getJPI.) Different sparsity-exploiting methods are possible; the default sparse method is direct determination using bicoloring [Coleman and Verma 1998a].

`[f,J]=evalJ(fun,x,Extra,m,JPI,verb)` Indicates the display level.
`verb` ≤ 0 No display.
`verb` ≥ 1 The number of groups used are displayed.
`verb` ≥ 2 Information is displayed in graph form.

`[f,J]=evalJ(fun,x,Extra,m,JPI,verb,fdstep)` Scalar `fdstep` denotes the finite-difference step size, for use when `method = 'f'`(see getJPI).

evalH

Purpose.

Compute the value of a scalar-valued function, the gradient, and possibly the Hessian matrix. When the Hessian matrix is computed, sparsity is exploited (using graph-coloring techniques, etc. [Coleman and Cai 1986; Coleman and Moré 1984a])

Synopsis.

```
v=evalH(fun,x)

v=evalH(fun,x,Extra)

[v,grad]=evalH(fun,x,Extra)

[v,grad,H]=evalH(fun,x,Extra,HPI)

[v,grad,H]=evalH(fun,x,Extra,HPI,verb)

[v,grad,H]=evalH(fun,x,Extra,HPI,verb,fdstep)
```

Description.

`[v,grad]=evalH(fun,x,Extra)` Determine the (scalar) value and gradient (dense vector) of `fun` at the input argument `x`. The first input argument, `fun`, is an integer handle identifying the user function. You can provide a *full* matrix, `Extra`, to be used by your target function. `Extra` cannot be a MATLAB *sparse* matrix.

`[v,grad,H]=evalH(fun,x,Extra,HPI)` Evaluate the sparse Hessian matrix `H` at `x`. `HPI` encodes the "coloring" information about `H` required to compute a compact representation of `H`. (See `getHPI`.) Different sparsity-exploiting methods are possible; the default sparse method used is direct determination (ignoring the symmetry).

`[v,grad,H]=evalH(fun,x,Extra,HPI,verb,fdstep)` Scalar `fdstep` denotes the finite-difference step size, for use when the finite-differencing option is selected (see `getHPI`).

getJPI

Purpose.

Compute sparsity and coloring information to allow for the efficient determination of a (sparse) Jacobian matrix.

Synopsis.

```
JPI= getJPI(fun, m)
JPI= getJPI(fun, m, n)
JPI= getJPI(fun, m, n,Extra)
JPI= getJPI(fun, m, n, Extra, method)
JPI= getJPI([], m, n, Extra, method, SPJ)
```

Description.

`JPI= getJPI(fun, m, n, Extra)` encapsulates (in a MATLAB sparse matrix) the sparsity pattern and graph-coloring information necessary to efficiently compute the sparse Jacobian matrix; the coloring determined corresponds to the default—direct bicoloring. The Jacobian matrix is assumed to be `m` × `n`. You can provide a *full* matrix, `Extra`, to be used by your target function `fun`.

`JPI= getJPI(fun, m, n,Extra, method)` overrides the default coloring.

`method = 'd'`: direct bicoloring (the default).
`method = 's'`: substitution bicoloring.
`method = 'c'`: one-sided column method.
`method = 'r'`: one-sided row method.
`method = 'f'`: sparse finite difference.

`JPI= getJPI([], m, n,Extra, method, SPJ)` You can supply `SPJ`, a sparse MATLAB matrix representing the sparsity structure of the Jacobian matrix. The sparse matrix structure `SPJ` is required on input when `method = 'f'`.

getHPI

Purpose.

Compute the sparsity structure and graph-coloring information for the sparse Hessian matrix `H`.

Synopsis.

```
HPI= getHPI(fun, n)
HPI= getHPI(fun, n, Extra)
HPI= getHPI(fun, n, Extra, method)
```

`HPI= getHPI([], n, Extra, method, SPH)`

Description.

`HPI= getHPI(fun, n, Extra)` The sparsity structure and relevant coloring information (to allow for efficient calculation of the sparse Hessian `H`) is encapsulated in `HPI`, a sparse matrix. The default coloring corresponds to direct determination. You can provide a *full* matrix, `Extra`, to be used by your target function (if required).

`HPI= getHPI(fun, n,Extra,method)` overrides the default coloring.
`method = 'i-a'`: The default, ignore the symmetry. Compute exactly using AD.
`method = 'd-a'`: direct method [Coleman and Moré 1984a], using AD.
`method = 's-a'`: substitution method [Coleman and Cai 1986] using AD.
`method = 'i-f'`: ignore the symmetry and use finite differences (FD)
`method = 'd-f'`: direct method [Coleman and Moré 1984a] with FD.
`method = 's-f'`: substitution method [Coleman and Cai 1986] with FD.

`HPI= getHPI(fun, n,Extra,method,SPH)` `SPH` is a sparse MATLAB matrix representing the sparsity structure of the Hessian matrix. The sparse structure `SPH` is required as input when `method = 's-f'`.

B. THE AD TOOL DRIVERS

Additional functions for driving the plug-in AD tool are described in this section. These drivers are all in form of MEX files.

forwprod

Purpose.

Computes the Jacobian matrix product, $J \times V$, where J is the Jacobian of a nonlinear vector mapping and V is a matrix. The product is computed directly via automatic differentiation—the cost is proportional to the number of columns in V. Note: `forwprod` is particularly efficient when the number of columns of V is small. Otherwise, when J is sparse it may be more efficient to compute J first (using `evalJ` and exploiting sparsity) and then perform the multiplication.

Synopsis.

```
[f,JV]=forwprod(fun,x,V)
[f,JV]=forwprod(fun,x,V,m)
[f,JV]=forwprod(fun,x,V,m,Extra)
```

Description.

`[f,JV]=forwprod(fun,x,V,m,Extra)` returns the function value and the product `JV` $= J *$ `V`, evaluated at `x`. The row dimension of the

Jacobian matrix is `m`. You can provide a full matrix, `Extra`, for use in your target function `fun` (if required).

revprod

Purpose.

Compute $W^T \times J$ where $J = J(x)$ is the Jacobian matrix of a nonlinear vector mapping and W is an arbitrary (consistent) matrix. The product is computed directly via automatic differentiation with the computational cost proportional to the number of columns in W. Note: `revprod` is particularly efficient when the number of columns of W is small. Otherwise, when J is sparse it may be more efficient to compute J first (using `evalJ` and exploiting sparsity) and then perform the multiplication.

Synopsis.

```
[f,WJ]=revprod(fun,x,W);
```

```
[f,WJ]=revprod(fun,x,W,Extra);
```

Description.

`[f,WJ]=revprod(fun,x,W,Extra)` returns the function value and the product `WJ` $= (W^T * J)^T = J^T W$. A full matrix, `Extra`, can be provided to be used by a target function `fun` (if required).

HtimesV

Purpose.

Compute $H \times V$ where $H = H(x)$ is a Hessian matrix of a scalar-valued function and V is a compatible matrix. Note: Function `HtimesV` is particularly efficient when the number of columns of V is small. Otherwise, when H is sparse it may be more efficient to compute H first (using `evalH` and exploiting sparsity) and then perform the multiplication.

Synopsis.

```
HV=HtimesV(fun,x,V)
```

```
HV=HtimesV(fun,x,V,Extra)
```

Description.

`HV=HtimesV(fun,x,V,Extra)` returns the product `HV` $= H * V$, where the Hessian matrix H is evaluated at the given point `x`. You can provide a full matrix, `Extra`, to be used (if required) by your target function `fun`.

hesssp

Purpose.

Computes the sparsity pattern of the Hessian matrix.

Synopsis.

```
SPH=hesssp(fun,n)
```

```
SPH=hesssp(fun,n,Extra)
```

Description.

`SPH=hesssp(fun,n,Extra)` returns the `n` × `n` sparsity structure of the Hessian matrix. `SPH` is a MATLAB sparse matrix. Note the current point is not required: a superstructure of the sparsity structure for all points x is returned. The structure can be displayed by `spy(SPH)`. A full matrix, `Extra`, can be provided to be used by your target function `fun`, if required.

jacsp

Purpose.

Compute the sparsity pattern of the Jacobian matrix.

Synopsis.

```
SPJ=jacsp(fun,m)
```

```
SPJ=jacsp(fun,m,n)
```

```
SPJ=jacsp(fun,m,n,Extra)
```

Description.

`SPJ=jacsp(fun,m,n,Extra)` returns the `m` × `n` sparsity structure of the Jacobian matrix. `SPJ` is a MATLAB sparse matrix. A full matrix, `Extra`, can be provided to be used by a target function `fun`.

REFERENCES

AVERICK, B. M., MORÉ, J. J., BISCHOF, C. H., CARLE, A., AND GRIEWANK, A. 1994. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput. 15*, 2 (Mar. 1994), 285–294.

BERZ, M., BISCHOF, C., CORLISS, G., AND GRIEWANK, A., Eds. 1996. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, PA.

BISCHOF, C. H., BOUARICHA, A., KHADEMI, P. M., AND MORÉ, J. J. 1997. Computing gradients in large-scale optimization using automatic differentiation. *INFORMS J. Comput. 9*, 2, 185–194.

COLEMAN, T. F AND CAI, J. Y. 1986. The cyclic coloring problem and estimation of spare hessian matrices. *SIAM J. Algebr. Discret. Methods 7*, 2 (Apr. 1986), 221–235.

COLEMAN, T. F. AND JONSSON, G. 1999. The efficient calculation of structured gradients using automatic differentiation. *SIAM J. Sci. Comput. 20*, 4, 1430–1437.

COLEMAN, T. F. AND MORÉ, J. J. 1974. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Program. 28*, 243–270.

COLEMAN, T. F. AND MORÉ, J. J. 1983. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal. 20*, 187–209.

COLEMAN, T. F. AND VERMA, A. 1996. Structure and efficient Jacobian calculation. In *Computational Differentiation: Techniques, Applications, and Tools*, M. Berz, C. Bischof, G. Corliss, and A. Griewank, Eds. SIAM, Philadelphia, PA, 149–159.

COLEMAN, T. F. AND VERMA, A. 1996. Structure and efficient Hessian calculation. In *Proceedings of the '96 International Conference on Advances in Nonlinear Programming*, Y. X. Yuan, Ed. Kluwer Academic, Dordrecht, Netherlands, 57–72.

COLEMAN, T. F. AND VERMA, A. 1997. ADMIT-1: Automatic differentiation and MATLAB interface toolbox, User guide. Tech. Rep. CTC97TR271. Theory Center, Cornell University, Ithaca, NY.

COLEMAN, T. F. AND VERMA, A. 1998a. ADMAT: An automatic differentiation toolbox for MATLAB. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-Operable Scientific and Engineering Computing*, SIAM, Philadelphia, PA.

COLEMAN, T. F. AND VERMA, A. 1998b. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM J. Sci. Comput. 19*, 4, 1210–1233.

COLEMAN, T. F. AND VERMA, A. 1999. ADMIT-2: Automatic differentiation and MATLAB interface toolbox for structured computation, User guide. Tech. Rep. Theory Center, Cornell University, Ithaca, NY. In preparation.

COLEMAN, T. F., GARBOW, B. S., AND MORE, J. J. 1984. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Softw. 10*, 3 (Sept.), 329–345.

COLEMAN, T. F., GARBOW, B. S., AND MORÉ, J. J. 1985. Software for estimating sparse Hessian matrices. *ACM Trans. Math. Softw. 11*, 4 (Dec.), 363–377.

CURTIS, A. R., POWELL, M. J. D., AND REID, J. K. 1974. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl. 13*, 117–119.

GAREY, M. AND JOHNSON, D. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., New York, NY.

GRIEWANK, A. 1993. Some bounds on the complexity of gradients. In *Complexity in Nonlinear Optimization*, P. Pardalos, Ed. World Scientific Publishing Co., Inc., River Edge, NJ, 128–161.

GRIEWANK, A. 1994. Tutorial on computational differentiation and optimization. In *Proceedings of the 15th International Mathematical Programming Symposium*, University of Michigan, Ann Arbor, MI.

GRIEWANK, A. AND CORLISS, G. F., Eds. 1991. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, PA.

GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Softw. 22*, 2, 131–167.

HOSSAIN, A. K. M. AND STEIHAUG, T. 1995. Computing a sparse Jacobian matrix by rows and columns. Tech. Rep. 109. University of Bergen, Bergen, Norway.

NEWSAM, G. N. AND RAMSDELL, J. D. 1983. Estimation of Sparse Jacobian Matrices. *SIAM J. Algebr. Discret. Methods 4*, 404–417.

POWELL, M. J. D. AND THOINT, PH. L. 1979. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal. 16*, 1060–1074.

RICH, L. C. AND HILL, D. R. 1992. Automatic differentiation in MATLAB. *Appl. Numer. Math. 9*, 1 (Jan. 1992), 33–43.

Received: January 1998; revised: June 1999 and November 1999; accepted: December 1999