

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 paper provides an introduction to the design and usage of ADMIT-1.

General Terms: Numerical Software, MATLAB

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

1. INTRODUCTION

The efficient numerical solution of nonlinear systems of algebraic equations $F(x) = 0$, $F(x) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, usually requires the repeated calculation or estimation of the matrix of first derivatives, the Jacobian matrix, $J(x) \in \mathbb{R}^{m \times n}$. 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 .

Similarly, the efficient numerical solution of numerical optimization problems involving a scalar valued function, $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, may require repeated computation

Authors' address: Computer Science Department and Cornell Theory Center, Cornell University, Ithaca NY 14850.

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works, requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM Inc., 1515 Broadway, New York, NY 10036 USA, fax +1 (212) 869-0481, or permissions@acm.org.
©1999 ACM

of the second derivative Hessian matrix $H(x) \in \mathfrak{R}^{n \times n}$. 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 paper 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 §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 [12]. There has been an earlier implementation of AD in MATLAB [25] (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 [20; 18; 19; 2]. We present a basic review of automatic differentiation in §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 [11; 10] 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 [7]. The software for structure computations is presented as a separate MATLAB toolbox, ADMIT-2 [15], an extension of the ADMIT-1 toolbox.

This paper is outlined as follows: In §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 §3. In §4 we present the software design of ADMIT-1 tool. In §5, we present a detailed usage of ADMIT-1 in a nonlinear equation solution using the Newton step. In §6 we present the algorithms and numerical results. In §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 online on WWW at the URL : <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 [18].

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

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

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

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

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

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

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

There are a number of cases when the elementary function is not differentiable (e.g. $f_{\text{elem}}^k(z_i) = \text{abs}(z_i)$ or $f_{\text{elem}}^k(z_i, z_j) = \text{max}(z_i, z_j)$). Sophisticated heuristic techniques are developed to treat these cases. For more details consult [18].

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 $z_k = f_{\text{elem}}^k(z_i, z_j)$ the intermediate derivative, $\frac{dz_k}{dx}$ can be propagated in the forward mode as:

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

This chain rule based computation is done for all the intermediate variables z and for the output variables u , finally yielding the derivative $\frac{du}{dx}$.

The reverse mode computes the derivatives $\frac{du}{dz_k}$ for all intermediate variables backwards (i.e., in the reverse order) through the computation. For example, for the elementary step $z_k = f_{\text{elem}}^k(z_i, z_j)$, the derivatives are propagated as:

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

At the end of computation of the reverse mode the derivative $\frac{du}{dx}$ 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 $J^T v$ given a vector v , where J is the Jacobian of a nonlinear mapping [18]. 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 [14; 9; 8; 4].

Sparse finite differencing techniques were first introduced by Curtis, Powell and Reid [16]; Coleman and Moré [9; 8; 4] and Newsam and Ramsdell [23] 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 [14; 1; 3].

3.1 Computation of a sparse Jacobian

One way to approach the problem of estimating a sparse Jacobian matrix of a mapping $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$, is in the following terms : given a sparse $m \times n$ matrix J , obtain vectors d_1, d_2, \dots, d_p such that the products Jd_1, Jd_2, \dots, 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 $d_1 = e_1 + e_4 + e_7 + \dots, d_2 = e_2 + e_5 + e_8 + \dots, d_3 = e_3 + e_6 + e_9 + \dots$. 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 [9; 5; 16]. The alternative row approach can be phrased: obtain vectors d_1, d_2, \dots, d_p such that the products $J^T d_1, J^T d_2, \dots, J^T d_p$ determine J uniquely. The vectors d_i determined completely by the non-zero 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 bi-coloring approach [14], 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 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 done by Hossian and Steihaug [22].

The motivation for taking this 2-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.

Bi-coloring circumvents this problem, and is never worse than 1-sided coloring. Here is a simple example which demonstrates the advantage of bi-coloring. Consider

the following n -by- n Jacobian, symmetric in structure but not in value:

$$J = \begin{pmatrix} \square & \triangle & \triangle & \triangle & \triangle \\ \square & \diamond & & & \\ \square & & \diamond & & \\ \square & & & \diamond & \\ \square & & & & \diamond \end{pmatrix}. \quad (1)$$

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. Therefore, if matrix W corresponds to a “consistent row partition” then W has n rows and the work to evaluate $W^T J$ by the reverse mode of AD is proportional to $n \cdot \omega(F)$. In this example the use of a bi-coloring 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 i^{th} 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 $W^T J$.

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 [14; 17]; therefore, heuristic schemes are considered to construct the “bi-partition”. For more insight into the problem involved and algorithmic details, refer to §6.

Below are performance results obtained for bi-coloring summarized from [14]: Table 1 shows the summary of the performance of bi-coloring on a linear programming testbed of matrices; Table 2 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 $J^T d$) needed to compute the sparse Jacobian matrices in the collection.

| Bi-coloring | | 1-sided Coloring | |
|-------------|--------------|------------------|-----|
| Direct | Substitution | column | row |
| 337 | 270 | 1753 | 452 |

Table 1. Totals for LP Collection (<http://www.netlib.org/lp/data/>)

Also, similar to the results reported in [1] for forward-mode direct determination, the Jacobian matrices determined by our bi-coloring/AD approach are significantly and uniformly more accurate than the finite-difference approximations

| Bi-coloring | | 1-sided Coloring | |
|-------------|--------------|------------------|-----|
| Direct | Substitution | column | row |
| 320 | 244 | 732 | 738 |

Table 2. Totals for Harwell-Boeing Collection (ftp from orion.cerfacs.fr)

(usually around 6 digits more 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 [14].

In §6, we present an overview of implementation of the one-sided column and row methods and the bi-coloring 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, \dots, d_p are needed to compute all the nonzeros of $\nabla^2 f(x)$ using the products $\nabla^2 f(x)d_1, \nabla^2 f(x)d_2, \dots, \nabla^2 f(x)d_p$.

The algorithms that we have implemented are based on the work of Powell and Toint [24], and Coleman and Moré [4; 8; 5]. 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 [4].

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 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 are denoted by $\chi(G^2)$. The intersection graphs and their construction are explained in detail in §6.
- (2) **Direct – exploiting symmetry** : This is the path-coloring method as described in [8]. 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 [4]. The complexity in this case is given by the 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 §6 which illustrates the selection of any of above three methods for computing a Hessian matrix.

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 §4.1, is required.

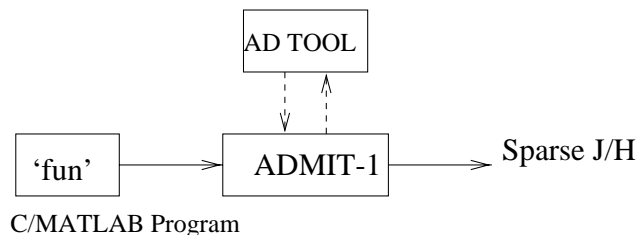


Fig. 1. Design of ADMIT-1 toolbox

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 §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 plugged into ADMIT-1, but cannot take advantage of the bi-coloring 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 §5.

The five functionality features expected from the AD tool are listed below. The function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is a scalar-valued function and $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ is a vector-valued function.

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

- (2) **Matrix-Jacobian (reverse) product.** $(F, x, W) \rightarrow J(x)^T W$.
- (3) **Jacobian Sparsity Pattern.** $F \rightarrow SPJ$.
- (4) **Hessian-Matrix product.** $(f, x, V) \rightarrow H(x)V$.
- (5) **Hessian Sparsity Pattern.** $f \rightarrow SPH$.

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 $= \nabla f(x) = J^T$.

The packages ADOL-C [21] and ADMAT [13] satisfy the requirements listed above.

The sparse Jacobian and Hessian matrices can be computed using a method of your choice, e.g., bi-coloring. 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 bi-coloring technique is replaced with the 1-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 NONLINEAR EQUATION SOLUTION

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 1-dimensional array corresponding to a 2-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 $y = F(x)$, $F : \mathfrak{R}^n \rightarrow \mathfrak{R}^n$ where

$$y(1) = 2x(1)^2 + \sum_1^n x(i)^2,$$

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

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

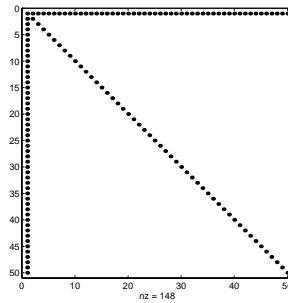


Fig. 2. The sparsity structure of Jacobian J

ADMIT-1 when using ADOL-C as the plug-in AD tool 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<m;j++)
        y[j]=x[j]*x[j];
    for(j=0;j<m;j++)
    {
        /* Dense first row */
        y[0]=y[0]+x[j]*x[j];
        /* Dense first column */
        y[j]=y[j]+x[0]*x[0];
    }
}
    
```

To evaluate the function F and the Jacobian J at $x' = (1, 1, \dots, 1)$ for $n = 5$:

```

>> 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 §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;

% 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 §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 [9]. 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 3.

| 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 bi-coloring method | $\chi_d(J)$ |
| Substitution bi-coloring method | $\chi_s(J)$ |

Table 3. Various Methods for computing sparse Jacobian matrices

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 :

$$\chi_s(J) \leq \chi_d(J) \leq \min(\chi_c(J), \chi_r(J)) \quad (2)$$

Inequality (2) holds since bi-coloring subsumes both one-sided coloring techniques; for more details, refer to [9].

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 column intersection graph of the Jacobian sparsity structure. For a detailed treatment on this subject, please refer to Coleman and Moré [9]. 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

```

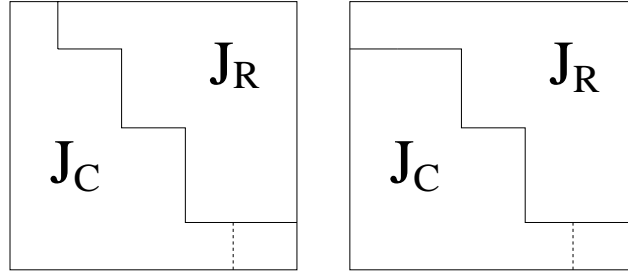
In the above code, J denotes the sparsity structure of the Jacobian matrix. Two columns are said to “intersect” if they both have a nonzero in the same row position. The main step of the algorithm consists of assigning each vertex in turn the lowest numbered color not yet used by the neighbours. The order in which the candidate columns are searched for is unspecified in the algorithm given above. Ordering based on graph coloring heuristics have proven to be effective [5]. 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 Bi-coloring 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 : $\bar{J} = P \cdot J \cdot Q = [J_C | J_R]$, as indicated in Figure 3. The construction of this 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 nonzeros 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 bi-coloring. For the algorithmic details, refer to [14]. Once the intersection graphs are colored, the boolean matrices


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

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.

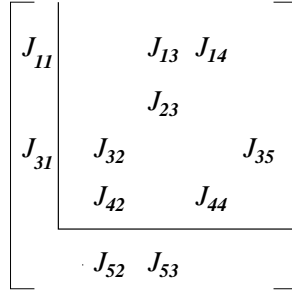


Fig. 4. Example Partition

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

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

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

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

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 $(W^T J, JV)$ by a substitution process. This is evident from the illustrations in Figure 5.

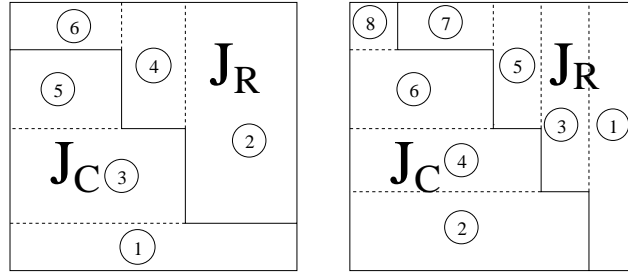


Fig. 5. Substitution Orderings

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) and 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 $JV, W^T J$:

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

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

It is now easy to verify that all nonzeros 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 k^{th} 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:

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

where $\rho(A)$ is the maximum number of nonzeros 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 nonzeros in M
- (3) Find $c \in C$ with fewest nonzeros in M
- (4) Repeat Until $M = \emptyset$
 - if $\rho(J_R^T) + \max(\rho(J_C), \text{nnz}(r)) < (\rho(J_C) + \max(\rho(J_R^T), \text{nnz}(c)))$ (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, upon completion, J_R, J_C 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 bi-coloring technique. The test function $F \in \mathfrak{R}^{n \times n}$ 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 [14]. Our results, shown in Figure 6, suggest the following order of execution time requirement by different techniques for the given test function:

$$FD > AD/row > AD/column > AD/bi-coloring(direct) > AD/bi-coloring(substitution).$$

In general, the above order will vary a little depending on the problem you are solving, e.g. for the special case of computing gradient, typically the AD/row method will be quicker than $AD/column$. 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 1 and 2.

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.

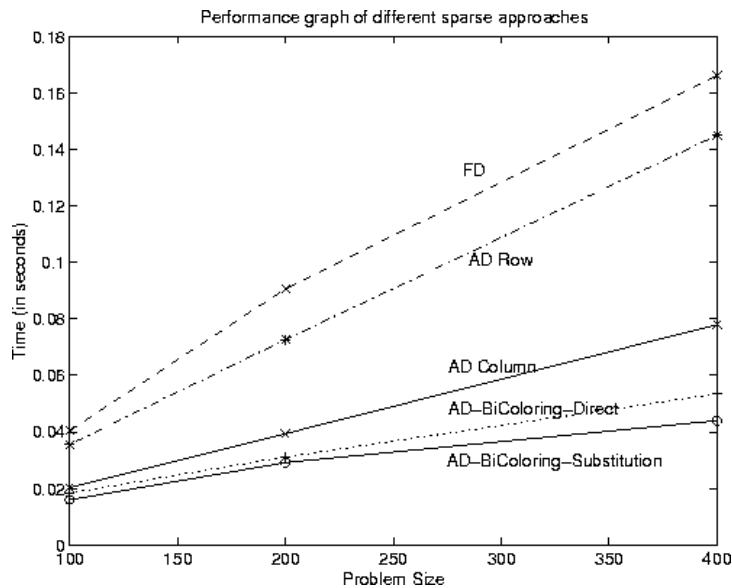


Fig. 6. A comparison of different sparse techniques

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:

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

6.2 Algorithms for computing sparse Hessians

The algorithms we have implemented for this step are based on the work of Powell and Toint [24] and Coleman and Moré [6].

- (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 same as the one used for 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 [6] :

Path coloring algorithm

Let $G = (V, E)$ be the adjacency graph.

for $k = 1, 2, \dots$

- (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 [4]. In summary, the algorithm involves finding a permutation π , such that 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, using the symmetric minimum degree works good) π , such that the column intersection graph of L_π 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;
>> JPI_d = getJPI(fun,m,n); ← JPI for direct bi-coloring (default) method
>> JPI_s = getJPI(fun,m,n,[],'s'); ← JPI for substitution bi-coloring method
>> JPI_c = 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:

```

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

```

Similarly for Hessians :

```

>> n=100;
>> [HPI i,SPH] = getHPI(fun,n); ← HPI for ignore symmetry (default) method
>> HPI d = 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 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 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 the MEX files since it needs the full MATLAB source code to work).

REFERENCES

- [1] B. M. Averick, J. J. Moré, C. H. Bischof, A. Carle, and A. Griewank, *Computing large sparse Jacobian matrices using automatic differentiation*, SIAM Journal on Scientific Computing, 15 (1994), pp. 285–294.
- [2] M. BERZ, C. BISCHOF, G. CORLISS, AND A. GRIEWANK, eds., *Computational Differentiation: Techniques, Applications, and Tools*, SIAM, Philadelphia, Penn., 1996.
- [3] C. H. Bischof, A. Bouaricha, P. M. Khademi, and J. J. Moré, *Computing gradients in large-scale optimization using automatic differentiation*, INFORMS J. Computing, 9 (1997), pp. 185–194.
- [4] T. F. Coleman and J. Y. Cai, *The cyclic coloring problem and estimation of sparse Hessian matrices*, SIAM J. Alg. Disc. Meth., 7 (1986), pp. 221–235.
- [5] T. F. Coleman, B. S. Garbow, and J. J. Moré, *Software for estimating sparse Jacobian matrices*, ACM Trans. Math. Software, 10 (1984), pp. 329–345.
- [6] ———, *Software for estimating sparse Hessian matrices*, ACM Trans. Math. Software, 11 (1985), pp. 363–377.
- [7] T. F. Coleman and G. Jonsson, *The efficient calculation of structured gradients using automatic differentiation*, SIAM Journal on Scientific Computing, 20 (1999), pp. 1430–1437.
- [8] T. F. Coleman and J. J. Moré, *Estimation of sparse Hessian matrices and graph coloring problems*, Math. Programming, 28 (1984), pp. 243–270.
- [9] ———, *Estimation of sparse Jacobian matrices and graph coloring problems*, SIAM J. on Numerical Analysis, 20 (1984), pp. 187–209.

- [10] T. F. Coleman and A. Verma, *Structure and efficient Hessian calculation*, in Advances in Nonlinear programming, proceedings of 96 International conference on nonlinear programming, Y. xiang Yuan, ed., Kluwer Academic Publishers, Boston/Dordrecht/London, 1996, pp. 57–72.
- [11] ———, *Structure and efficient Jacobian calculation*, in Computational Differentiation: Techniques, Applications, and Tools, M. Berz, C. Bischof, G. Corliss, and A. Griewank, eds., SIAM, Philadelphia, Penn., 1996, pp. 149–159.
- [12] ———, *ADMIT-1: Automatic differentiation and matlab interface toolbox, User Guide*, Tech. Rep. CTC97TR271, Theory Center, Cornell University, 1997.
- [13] ———, *ADMAT: An automatic differentiation toolbox for matlab*, in Proceedings of SIAM workshop on Object Oriented Methods for Inter-operable scientific and Engineering Computing, Philadelphia, PA, 1998, SIAM.
- [14] ———, *The efficient computation of sparse Jacobian matrices using automatic differentiation*, SIAM Journal on Scientific Computing, 19 (1998), pp. 1210–1233.
- [15] ———, *ADMIT-2: Automatic differentiation and matlab interface toolbox for structured computation, User Guide*, tech. rep., in preparation, 1999.
- [16] A. R. Curtis, M. J. D. Powell, and J. K. Reid, *On the estimation of sparse Jacobian matrices*, J. Inst. Math. Appl., 13 (1974), pp. 117–119.
- [17] M. R. Garey and D. S. Johnson, *Computers and intractability, A guide to the theory of NP-completeness*, W.H. Freeman, San Francisco, 1979.
- [18] A. Griewank, *Some bounds on the complexity of gradients, Jacobians, and Hessians*, in Complexity in Nonlinear Optimization, P. Pardalos, ed., World Scientific Publishers, 1993, pp. 128–161.
- [19] ———, *Tutorial on computational differentiation and optimization*, in Proceedings of XV International Mathematical Programming Symposium, Ann Arbor, MI, 1994, University of Michigan.
- [20] A. GRIEWANK AND G. F. CORLISS, eds., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, Penn., 1991.
- [21] A. Griewank, D. Juedes, and J. Utke, *ADOL-C, a package for the automatic differentiation of algorithms written in C/C++*, ACM Trans. On Math. Software, 22 (1996), pp. 131–167.
- [22] A. K. M. Hossain and T. Steihaug, *Computing a sparse Jacobian matrix by rows and columns*, Technical Report 109, Department of Informatics, University of Bergen,, Bergen, June 1995.
- [23] G. N. Newsam and J. D. Ramsdell, *Estimation of sparse jacobian matrices*, SIAM J. Alg. Disc. Meth., 4 (1983), pp. 404–417.
- [24] M. J. D. Powell and P. L. Toint, *On the estimation of sparse Hessian matrices*, SIAM J. Numer. Anal., 16 (1979), pp. 1060–1074.
- [25] L. Rich and D. Hill, *Automatic differentiation in MATLAB*, Applied Numerical Mathematics, 9 (1992), pp. 33–43.

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., $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$.

`[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 bi-coloring [14].

`[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. [8; 4])

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 bi-coloring. The Jacobian matrix is assumed to be $m \times n$. 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 bi-coloring (the default).
method = 's': substitution bi-coloring.
```

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 [8], using AD.

method = 's-a': substitution method [4] using AD.

method = 'i-f': ignore the symmetry and use finite differences(FD)

method = 'd-f': direct method [8] with FD.

method = 's-f': substitution method [4] with FD.

`HPI= getHPI(fun, n,Extra,method,SPH)` You can supply `SPH`, 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 $WJ = (W^T * J)^T = J^T W$. You can provide a full matrix, `Extra`, to be used by your 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 $n \times n$ 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)`. You can provide a full matrix, `Extra`, 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 $m \times n$ sparsity structure of the Jacobian matrix. `SPJ` is a MATLAB sparse matrix. You can provide a full matrix, `Extra`, to be used by your target function “`fun`”.