Cornell Computer Science Technical Report TR 98-1663 ADMIT-1 : Automatic Differentiation and MATLAB Interface Toolbox *

Thomas F. Coleman[†] Arun Verma[‡]

January 9, 1998

Abstract

ADMIT-1 enables you to compute *sparse* Jacobian and Hessian matrices, using automatic differentiation technology, from a MATLAB environment. You need only supply a function to be differentiated and ADMIT-1 will exploit sparsity if present to yield sparse derivative matrices (in sparse MATLAB form). A generic AD 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.

1 Introduction

The efficient numerical solution of nonlinear systems of algebraic equations $F(x) = 0$, $F(x) : \Re^n \to \Re^m$, 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.

Similarly, the efficient numerical solution of numerical optimization problems involving a scalar valued function, $f(x) : \Re^n \to \Re$, 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 paper we present software to compute sparse Jacobian and Hessian matrices efficiently and painlessly using automatic differentiation 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 using ADMIT-1, refer to the ADMIT-1 user manual [?].

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. For a basic review of automatic differentiation, we refer the readers to [?, ?].

[†]Computer Science Department and Center for Applied Mathematics, Cornell University, Ithaca NY 14850.

[‡]Computer Science Department, Cornell University, Ithaca NY 14850.

*This research was partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-97ER25013.

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

This paper is outlined as follows: In §2, we review the sparsity exploiting techniques to compute the sparse Jacobian and Hessian matrices. The ADMIT-1 software and related information can be accessed online on WWW at the URL : http://www.cs.cornell.edu/home/verma/AD/research.html.

2 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 [?, ?, ?, ?].

Sparse finite differencing techniques were first introduced by Curtis, Powell and Reid [?]; Coleman and Moré [?, ?, ?] and Newsam and Ramsdell [?] 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 [?, ?, ?].

2.1 Computation of a sparse Jacobian

One way to approach the problem of estimating a sparse Jacobian matrix of a mapping $F : \Re^n \rightarrow \Re^m$, is in the following terms : given a sparse m by 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. This approach is called the one-sided column approach for computing a sparse Jacobian [?, ?, ?]. Finite differences can be used to approximate the products Jd; automatic differentiation in the forward mode can be used to compute the products Jd exactly.

The alternative row approach can be phrased: obtain vectors $d_1, d_2, \ldots, d_p$ such that the products $J^T d_1, J^T d_2, \ldots, J^T d_p$ determine J uniquely. 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 [?], 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 $(W^T J, JV)$. Given an arbitrary n-by-t_V matrix V, product JV can be directly calculated using automatic differentiation in the "forward mode"; given an arbitrary m-by-t_W matrix W, the product $W^T J$ can be calculated using automatic differentiation in the "reverse mode", e.g., [?, ?].

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 provably better than 1-sided coloring always. 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 both the direct determination problem and determination by substitution can be constructed based directly on Jacobian structure. The associated graph coloring problems are shown to be NP-complete [?, ?]; 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 : 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

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

TABLE 2

Totals for Harwell-Boeing Collection

Also, similar to the results reported in [?] 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. 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 [?].

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.

2.2 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 structure of $\nabla^2 f(x)$ is known, then usually a few well chosen directions $d_1, d_2, \ldots, d_p$ are needed to compute all the nonzeros of $\nabla^2 f(x)$ using the products $\nabla^2 f(x) d_1, \nabla^2 f(x) d_2, \ldots, \nabla^2 f(x) d_p$.

The algorithms that we have implemented are based on the work of Powell and Toint [?], and Coleman and Moré [?, ?, ?]. 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.

For a complete review on this subject, refer to Coleman and Cai [?]. 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. So $H_{i,j}$ and $H_{j,i}$ are computed independently. The minimum number of groups needed to compute the Hessian via this method is indicated by $\chi(G^2)$.
2. **Direct – exploiting symmetry :** This is the path-coloring method as described in [?]. The path coloring chromatic number is denoted by $\chi_\pi(G)$.
3. **Substitution – exploiting symmetry :** This is the cyclic-coloring method as described in [?]. The cyclic coloring chromatic number 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 set 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. For details on usage of these different methods, look under the function getHPI.

3 Software design of ADMIT-1

The design of the ADMIT-1 toolbox is as shown in figure 1. A generic AD tool, with functionality described in §4, is required.

The core of the ADMIT toolbox are two routines, evalJ and evalH. In §7 we describe the usage of these two functions. ADMIT-1 uses the sparse techniques for computation of sparse Jacobian and Hessians (and other derivative information like gradient and Jacobian matrix products etc.). Refer to the appendix to learn about additional functionality of ADMIT-1.

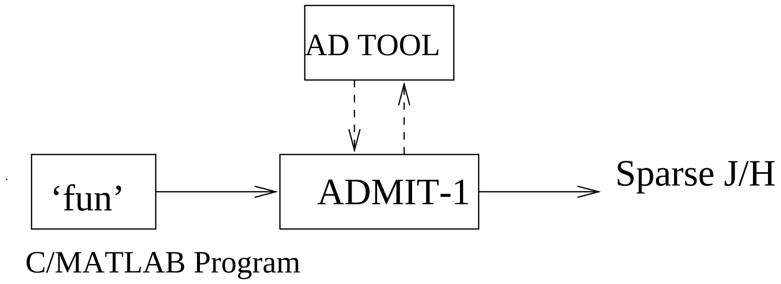

FIG. 1. *Design of ADMIT-1 toolbox*

4 Expected design of the underlying AD tool

The underlying AD plug-in tool should have implementation of both reverse and forward modes of automatic differentiation. In particular, the following five capabilities from the AD tool are needed in order to qualify as a plug-in tool for ADMIT-1.

The five functionality requirements from the AD tool are :

1. **Vector valued functions :**
   (a) **Jacobian-Matrix (forward) product**. $(F, x, V) \to J(x)V$.
   (b) **Matrix-Jacobian (reverse) product**. $(F, x, W) \to J(x)^T W$.
   (c) **Jacobian Sparsity Pattern**. $F \to SPJ$.
2. **Scalar valued functions :**
   (a) **Hessian-Matrix product**. $(f, x, V) \to H(x)V$.
   (b) **Hessian Sparsity Pattern**. $f \to SPH$.

SPJ and SPH stand for sparsity pattern of the Jacobian and Hessian matrix respectively. ADMIT-1 requires the AD plug-in tool to be able to compute these sparsity patterns (to be precise, a superset of sparsity pattern at all input points x to take care of accidental cancellations of dependence w.r.t certain input variables).

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

ADOL-C [?] and ADMAT [?] satisfy these requirements.

5 An example

Here is simple example illustrating how to use ADMIT-1 to calculate the Jacobian of the function $y = F(x)$, $F : \Re^n \to \Re^n$ 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 $n = 50$. ADMIT-1 (for use with ADOL-C) requires a C program to evaluate F:

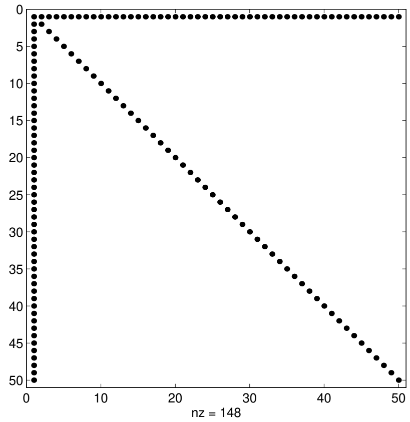

FIG. 2. *The sparsity structure of Jacobian J*

```
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];
    }
}
```

Assume this program is saved in file myfun.c. To evaluate the function F and the Jacobian J at $x' = (1, 1, ..., 1)$ for $n = 5$, and then display the structure of J:

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

      7
      2
      2
      2
      2
>> spy(J) ← Sparsity structure is displayed.
```

The second statement, involving "configf" and "getJPI", resolves the reference to the target function and extracts sparsity/coloring information respectively. As illustrated below in the Newton iteration example, only one execution of "configf" and "getJPI" is required for a given target function.

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.

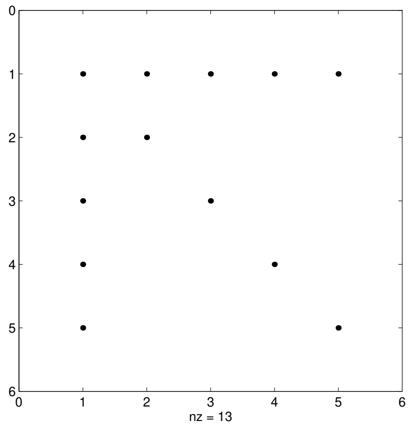

FIG. 3. *The sparsity structure of example computation*

6.1 Computing sparse Jacobians

There are basically five different options to compute a sparse Jacobian matrix using ADMIT-1. This is 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 method except that the former uses finite differences to approximate the product Jd. The various chromatic numbers satisfy the inequality :

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

These inequalities hold since bi-coloring subsumes both the single sided coloring techniques.

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 [?]. The method implemented in ADMIT-1 is outlined in the following pseudocode :

```
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
```

The algorithm presented above is in pseudo MATLAB code. Here 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 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 [?]. One such ordering, smallest degree ordering is the default ordering used in ADMIT-1.

The row one-sided 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 Here we present the algorithms for implementation of bi-coloring (both substitution and direct methods). The two combinatorial problems we face, corresponding to direct determination and determination by substitution, can both 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 4. The construction of this partition is crucial; however, we postpone that discussion until after we illustrate its utility. Assume $P = Q = I$ and $J = [J_C|J_R]$.

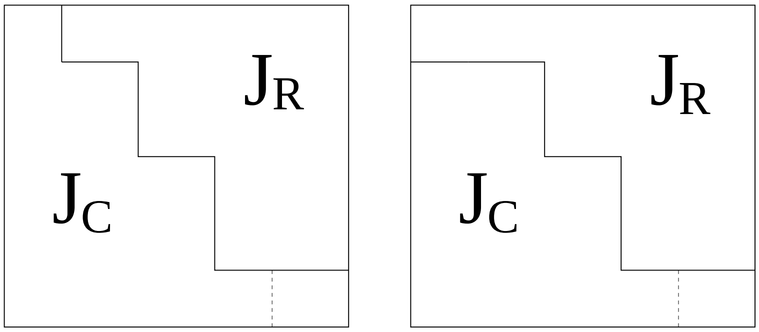

FIG. 4. *Possible partitions of the matrix* $\tilde{J} = P \cdot J \cdot Q$

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 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 [?]. 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 5 with the partition (J_C, J_R) shown.

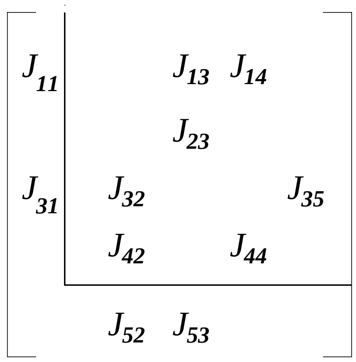

FIG. 5. *Example Partition*

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

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

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

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 6.

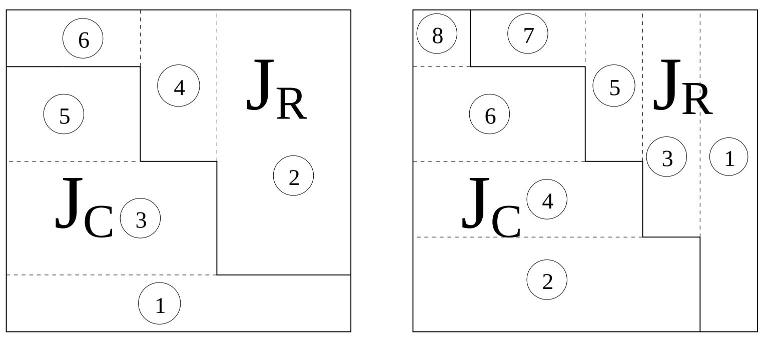

FIG. 6. *Substitution Orderings*

Figure 6 illustrates two of four possible nontrivial types of partitions. In both cases it is clear that nonzero elements in the section labeled "1" can be solved for directly – by the construction process they will be in different groups. 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 5.

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:

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

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

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 4.

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:

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

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 $M = \emptyset$

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

    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 reproduce some results here from [?] to illustrate the effectiveness of the bi-coloring technique. The test function F we use is a sample nonlinear function. Our results, reproduced in Figure 7, suggest the following order of execution time requirement by different techniques:

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

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. This bound is often loose in practise whereas $t_V \cdot \omega(F)$ is tight for finite-differencing since the subroutine to evaluate F is actually called (independently) t_V times.

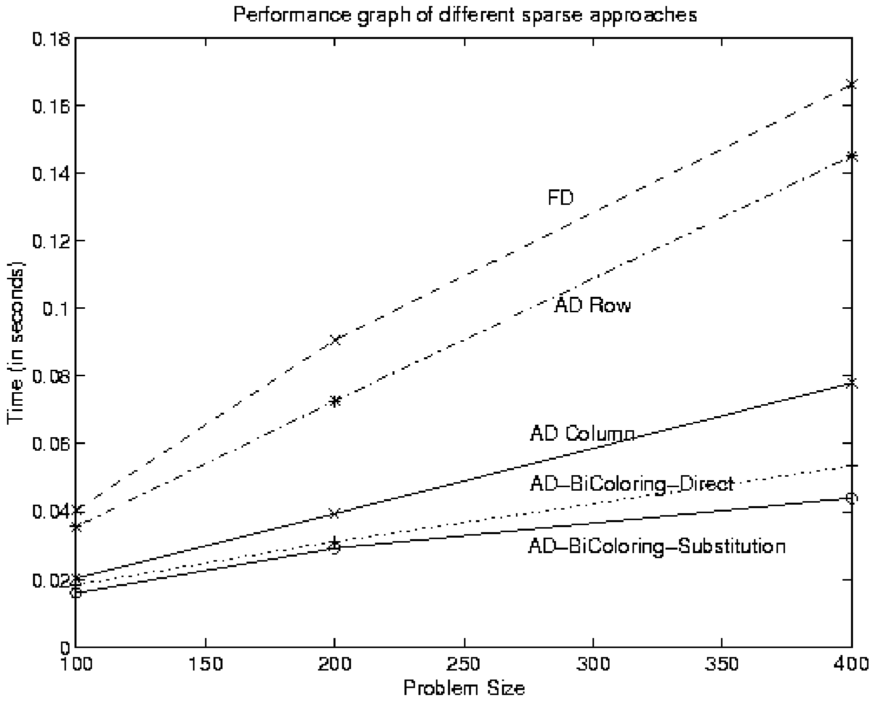

FIG. 7. *A comparison of different sparse techniques*

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 Figure 1, the column dimensions of V and W are equal, and the reverse mode is typically twice as more expensive as the forward mode. 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:

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

6.2 Algorithms for computing sparse Hessians

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

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 what is called the path coloring of the adjacency graph of the Hessian matrix. The algorithm involved can be found in detail in Coleman and Moré [?]. The algorithm can be spelled out like this :

   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

   In the end 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 [?]. 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.

All the three different methods have the same interface for finding the grouping and the permutation. Here we illustrate the ignhess method :

ignhess

Synopsis

g = ignhess(SPH)

[g,p]= ignhess(SPH)

Description

[g,p] = ignhess(SPH)

Returns the group assignment and the permutation ordering for computation of Hessian matrix by ignoring the symmetry altogether.

7 The ADMIT-1 functionality

In this section, we present the high level functionality of ADMIT-1. We describe the main functions evalJ and evalH which use the algorithms mentioned previously.

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 bi-coloring [?].

[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. [?, ?])

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 [?], using AD.

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

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

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

method = 's-f': substitution method [?] 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'.

8 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 bi-coloring (default) method
>> JPIs = getJPI(fun,m,n,[],'s'); ← JPI for substitution bi-coloring 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 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;
>> [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
```

9 Making your own "fun"

In this section 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.

```
void getfun(float* x,int n,float* y ,int m, float *Extra, int *numrows, int *numcols)
{

/* Crunch */

}
```

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)

% Crunch
```

9.1 An example function

This is a example C++ target function with an arrowhead Jacobian structure.

```
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];
    }
}
```

9.2 Configuring ADMIT-1 for your target C/C++ function

Use configf, from within MATLAB, to preprocess and compute the "handle" to your function.

For example suppose your target function is in file yourfun.c :

```
>> fun = configf('yourfun');
```

configf

Purpose:

Configures your C/C++ target function.

Synopsis

fun= configf(MainFileName)

fun= configf(MainFileName,SourceDir)

Description

fun= configf(MainFileName) Configures a C test problem in the file mainfilename.c. This file is assumed to be in the current working directory(the ADMIT-1 workspace).

fun= configf(MainFileName,SourceDir) Configures a C test problem in the directory SourceDir, and the top-level function is defined in file mainfilename.c Designed for test problems involving more than 1 file.

Note that for the MATLAB target functions, this configuration step is not needed and you can directly access ADMIT-1 functionality by using fun as the MATLAB file name.

Appendix

A Newton Process for nonlinear equations F(x)=0

We illustrate the use of ADMIT-1 in a Newton process. The example target function is the broyden nonlinear function. Here is the shell (MATLAB) program:

```
>> fun = 'broyd1a';
>> 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= broy1a(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
if (nargin < 5)
% Get the Coloring Info Once and for all
        JPI= getJPI(fun,n);
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);
```

B Newton step via conjugate gradient for nonlinear least squares(NLS)

It is easy to write a conjugate gradient solver for NLS using ADMIT-1. In this section, we describe the two conjugate gradient solvers in nonlinear least squares setting

$$min \quad \|F(x)\|_2^2 \tag{2}$$

Typically a Newton iteration is applied to solve (2), at least locally; two different kinds of Newton steps are popular :

- **Gauss Newton's step** : Solve for s.

$$J(x)^T J(x)s = -J(x)^T F(x) \tag{3}$$

- **Complete Newton step** : Solve for s.

$$H(x)s = -\nabla f(x) \tag{4}$$

where $f(x) := \|F(x)\|_2^2$.

IN ADMIT-1, we have conjugate gradient solvers, congrd and congrdH respectively for the Gauss-Newton step and the complete Newton step. Function congrd is described below.

congrd

Purpose:

Computes the Gauss Newton step $s = -(J(x)^T J(x))^{-1} J(x)^T F(x)$ via conjugate gradient method.

Synopsis

s=congrd(fun,x)
s=congrd(fun,x,m,tol)
s=congrd(fun,x,m,tol)
s=congrd(fun,x,m,tol,P)
s=congrd(fun,x,m,tol,P,J)
s=congrd(fun,x,m,tol,P,J,JPI)
[s,it]=congrd(fun,x,...)
[s,it,normres]=congrd(fun,x,...)

Description

s=congrd(fun,x,m,tol,P,J,JPI)
returns the step $s = -(J^T J)^{-1} J^T F$. The function F, denoted by the identifier fun, is assumed to be a $\Re^n \rightarrow \Re^m$ mapping. The user supplied tolerance tol on the residual norm is used. The user supplied preconditioner P is used. User can supply the Jacobian J(x), if not it is computed via AD. User can supply precomputed JPI, the partition and coloring information structure to increase efficiency.

[s,it] =congrd(fun,x,...) Number of iterations required are also returned.

[s,it,normres] =congrd(fun,x,...) normres contains the norm of the residual $Js - F$.

C 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".