# STRUCTURED AUTOMATIC DIFFERENTIATION

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Arun Verma
May 1998

STRUCTURED AUTOMATIC DIFFERENTIATION

Arun Verma, Ph.D.

Cornell University 1998

Differentiation is one of the fundamental problems in numerical mathemetics. The solution of many optimization problems and other applications require knowledge of the gradient, the Jacobian matrix, or the Hessian matrix of a given function.

Many large scale optimization applications (e.g., inverse problems) are very complex in nature. It becomes impractical to consider the function evaluation of such problems as a "black-box" function, since the computation is structured in some manner, going through a set of defined structured steps, i.e., problem structure. It pays to expose the problem structure in the computation to be able to compute the derivatives efficiently thus making the problem solution practical.

Automatic differentiation (AD) can compute fast and accurate derivatives of any degree computationally via propagating Taylor series coefficients using the chain rule. AD doesn't incur any truncation error and would yield exact results if the calculations were done in real arithmetic; in other words the derivatives obtained are accurate to machine precision.

This thesis is concerned with the efficient application of AD to large (and complex) optimization problems. The major theme is the structure exploitation of the

user problem. We present methodologies which allow AD to exploit problem structure. An important idea is the exploitation of sparsity in the Jacobian matrices: We present a scheme which combines the forward and reverse modes of AD.

Problem structure can be viewed in many different ways; one way is to look at the granularity of the operations involved. For example, differentiation carried out at the matrix-vector operations can lead to great savings in the time as well as space requirements. Figuring out the *kind* of computation is another way to view structure, e.g., partially separable or composite functions whose structure can be exploited to get performance gains. In this thesis we develop a general structure framework which can be viewed hierarchically and allows for structure exploitation at various levels. For example, for time integration schemes employing stencils it is possible to exploit structure at both the stencil level and the timestep level.

We also present some advanced structure exploitation ideas, e.g., parallelism in structured computations and using structure in implicit computations. The use of AD as a derivative computing engine naturally automates all the methodologies presented in this work – we present ways to make the design of numerical optimization software very transparent, and the presentation of problems by the user as easy as possible.

# Biographical Sketch

Arun grew up in northern India where he spent early years of his life and did his schooling. During these years Arun had developed a keen interest in mathematics and science & technology(also in music, among other fields, but he chose the former interest for his potential career) which led him to move to New Delhi to pursue a bachelor's degree in computer science from Indian Institute of Technology, New Delhi.

After completing 4 wonderful years at the IIT, he moved to Ithaca, New York in Fall 1994 and entered Cornell University to pursue his doctoral degree. He earned his M.S in Computer science in 1996, his Ph.D followed in 1998. He is continuing to pursue postdoctoral research in computer science at Cornell University.

*To my parents . . .*

# Acknowledgements

I would like to first thank my advisor, Tom Coleman for the constant support and invaluable guidance for this work. Tom provided me with unwavering encouragement and advice, while letting me free to explore and research on my own.

My thanks also go to my other committee members, Mike Todd and Steve Vavasis for the enthusiasm they showed in this work and the extensive comments they offered to improve the presentation of this thesis.

I would like to thank Fadil Santosa for introducing me to real-world inverse problems; the work reported in Chapter 6 is joint work with him. I wish to express my gratitude to Chris Bischof and his team at Argonne National Labs for providing me with valuable insight into this subject, when I worked at ANL in the summer of 1996. I thank my colleagues Yuying Li, Gudbjorn Jonsson and Adam Florence for all their help. I am also grateful to Helene Croft for her administrative support.

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-FG02-90ER25013 and in part by the Advanced Computing Research Institute, a unit of the Cornell Theory Center which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute.

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction and Background

The computational science field includes problems ranging from modeling physical phenomena, computation of option pricing in finance, and optimal control problems, to inverse problems in medical imaging and geosciences. The solution of problems in a variety of areas in computational science often involves presenting the problem as a numerical optimization problem which in turn requires computing derivatives of numerical functions. In numerical optimization derivatives, usually in the form of gradients, Jacobians and Hessians, are used to locate the extrema of a function; most optimization software includes some way of computing the derivatives (exactly or approximately) if not provided by the user.

In this thesis we advocate the use of automatic differentiation technology for computing derivatives in the context of large-scale optimization. A large part of this thesis deals with the applications themselves, understanding their structure and using the efficient application of automatic differentiation. The central theme of this work is to illustrate the interplay between the **automatic differentiation** technology, **large-scale nonlinear applications** and **numerical optimization software design**, resulting in development of efficient solution methods for a variety of applications.

## 1.1 Automatic Differentiation

Automatic differentiation is a chain-rule-based technique for evaluating the derivatives with respect to the input variables of functions defined by a high-level language computer program. AD relies on the fact that all computer programs, no matter how complicated, use a finite set of elementary (unary or binary, e.g., sin(.), sqrt(.)) operations as defined by the programming language. The value or 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 can be computed using the chain rule; this process is known as automatic differentiation [Gri90,Gri93].

A program computing the function $z = F(x), F : \Re^n \to \Re^m$, can be viewed as a sequence of scalar assignments $v_i = \psi_i < v_j >_{j \to i}$, where the relation $j \to i$ indicates that $v_j$ is computed before $v_i$. Hence, the vector $v$ can be thought of as a set of ordered variables such that $v_j, j \to i$, is computed before $v_i$ using the set of variables $\{v_k | k \to j\}$. Here $\psi_i$ represent elementary functions, which can be arithmetic operations and/or univariate transcendental functions.

Ordering the variables as above, we can partition the variables $v_j$ into three vectors:

$$\begin{aligned} x &\equiv (v_1, v_2, \ldots, v_n) \text{ (independent)} \\ y &\equiv (v_{n+1}, v_{n+2}, \ldots, v_{n+p}) \text{ (intermediate)} \\ z &\equiv (v_{n+p+1}, v_{n+p+2}, \ldots, v_{n+p+m}) \text{ (dependent).} \end{aligned}$$

In general, the number of intermediate variables is much larger than the dimensions of the problem, i.e., $p \gg m, n$.

Assume that all these elementary functions $\psi_i$ are well defined and have continuous elementary partials $c_{ij} = \frac{\partial \psi_i}{\partial v_j}, j < i$. Assuming without loss of generality that the dependent variables $z$ do not themselves occur as arguments of elementary functions, we can combine the partials $c_{ij}$ into the $(p+m) \times (n+p)$ matrix

$$C = (\, c_{\,n+i,j})_{1 \le i \le p+m}^{1 \le j \le n+p}$$

Unless elementary functions with more than two arguments are included in the library, each row of $C$ contains either one or two nonzero entries. We define a number[1]

$$q \equiv nnz(C) \le 2 \cdot (p+m) \approx 2p$$

Also, since the work involved in an elementary function is proportional to the number of arguments, it follows that

$$\omega(F) = \sum_{i=1}^{p+m} \omega(\psi_{n+i}) \sim q \tag{1.1}$$

where $\sim$ indicates proportionality and $\omega(\cdot)$ denotes the temporal complexity or the computational cost to carry out a certain operation.

Because of the ordering relation the square matrix $C$ is upper trapezoidal with $n-1$ nontrivial superdiagonals. Thus $C$ can be partitioned as

$$C = \left( \begin{array}{c|c} A & L \\ \hline B & M \end{array} \right) \tag{1.2}$$

where

$$A \in \Re^{p \times n}, B \in \Re^{m \times n}, L \in \Re^{p \times p}, M \in \Re^{m \times p},$$

and $L$ is strictly lower triangular. Application of the chain rule [Gri90] yields the Newton system:

[1] If $M$ is a matrix or a vector then "$nnz$(M)" is the **n**umber of **n**on**z**eroes in $M$.

$$\begin{aligned} A\Delta x + L\Delta y &= \Delta y \\ B\Delta x + M\Delta y &= -z \end{aligned}$$

or

$$\begin{pmatrix} A & L-I \\ B & M \end{pmatrix} \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} 0 \\ -z \end{pmatrix}. \tag{1.3}$$

If we eliminate the intermediate vector $\Delta y$ from (1.3), we get an expression (the Schur complement) for the Jacobian:

$$J \equiv B - M(L-I)^{-1}A = B + M(I-L)^{-1}A. \tag{1.4}$$

Since $I - L$ is a unit lower triangular matrix, the calculation of the matrix products $\tilde{A} \equiv (I-L)^{-1}A$ and $\tilde{M} \equiv M(I-L)^{-1}$ leads to two natural ways to compute $J$:

$$J = B + M\tilde{A} \quad \textit{or} \quad J = B + \tilde{M}A. \tag{1.5}$$

The alternative expressions for $J$ given in (1.5) define the two basic *modes* of automatic differentiation, *forward* and *reverse*.

- The *forward mode* corresponds to computing the rows of $\tilde{A}$ from $(I-L)\tilde{A} = A$, one by one, as the corresponding rows of $[A, L-I]$ are obtained from successive evaluation of elementary functions. Since this amounts to solutions of $n$ linear systems with lower-triangular matrix $[I-L]$, followed by multiplication of the dense columns of $\tilde{A}$ by $M$, the total computational effort is roughly $n \cdot q$ or $n \cdot \omega(F)$.

- The *reverse mode* corresponds to computing $\tilde{M}$ as solution to linear system $(I-L)^T\tilde{M}^T = M^T$. This back-substitution process can begin only after all elementary functions and their partial derivatives have been evaluated. Since this amounts to the solution of $m$ linear systems with lower triangular matrix $[I-L]$, followed by multiplication of the dense rows of $\tilde{M}$ by $A$, the total computational effort is roughly $m \cdot q$ or $m \cdot \omega(F)$.

We are interested in computing products of the form $JV$ and $W^TJ$. The product $JV$ can be computed using:

$$JV \quad = \quad BV + M[(I-L)^{-1}(AV)]$$

which can clearly be done in time proportional to $t_V \cdot \omega(F)$ when $V \in \Re^{n \times t_V}$. Analogously, the product $W^TJ$ can be computed by:

$$W^TJ \quad = \quad W^TB + [(W^TM)(I-L)^{-1}]A$$

which can be done in time proportional to $t_W \cdot \omega(F)$ assuming $W \in \Re^{m \times t_W}$.

## 1.2 Complexity

One key advantage of computational differentiation is that it allows an a priori bound on the cost of evaluating certain derivative objects in terms of the cost for evaluating the function itself. Consider a general nonlinear $F(x) : \Re^n \to \Re^m$. Let $\omega(\cdot)$ denote the temporal complexity or computational cost to carry out a certain operation and $S(\cdot)$ denotes the spatial (memory) complexity.

### Cost of basic forward and reverse mode

- Forward Mode: $(x, V \in \Re^{n \times p_1}) \to (F(x), JV)$
  - **Work cost:** $\omega(F, JV) = p_1 \cdot \omega(F)$
  - **space cost:** $S(F, JV) = S(F)$
- Reverse Mode: $(x, W \in \Re^{m \times p_2}) \to (F(x), J^T W)$
  - **Work cost:** $\omega(F, J^T W) = p_2 \cdot \omega(F)$
  - **space cost:** $S(F, J^T W) = numIvars(F)$

$numIvars(F)$ represents the total number of intermediate variables generated in the computation of $F$. All of these variables need to be saved for the reverse mode operation (equivalent to saving the matrix $L$ before you can start the back substitution) in the reverse order. In the Section 1.1 notation, $numIvars(F)$ is the same as $p$ which is also equal to the work involved in computing the function $F$, see equation 1.1.

One special case of the reverse mode is gradient evaluation, where we can use $W = 1$ (a scalar) to compute the gradient $\nabla f(x) \equiv J^T$ at a cost proportional to $\omega(F)$; the constant in front can be shown to be about 5 in practice. This is also known as the **cheap gradient result**. However, this does require space proportional to $numIvars(F)$ which can be prohibitive. Note however that the forward mode costs $n \cdot \omega(F)$. The efficiency of the forward mode evaluation of the gradient can be dramatically increased - i.e., the dependence on $n$ is removed - if $F$ has structure that can be exploited [BBKM95,CJ97].

Hence in the reverse mode

$$\omega(f, \nabla f) = 5 \cdot \omega(f), \quad S(f, \nabla f) = numIvars(f).$$

### Making reverse mode more effective

#### Space/time tradeoff

There are techniques by which the reverse mode can be made more effective in terms of space cost. For example, the checkpointing scheme by Andreas Griewank

[Gri92a] exploits the tradeoff involved in computational cost and space cost in reverse mode. The checkpointing scheme is a spectrum of methods ranging from linear checkpointing to multi-level (or partially recursive) checkpointing to fully recursive checkpointing. The fully recursive checkpointing method exploits the tradeoff to a maximum extent and achieves only a logarithmic growth of memory space requirements at the cost of a logarithmic growth in the time complexity, making the spatial complexity very reasonable and practical. The linear checkpointing schemes offer a different equation for tradeoff, where a linear reduction in spatial complexity is achieved at the cost of a linear increase in the computational cost.

**Fast reverse mode using structure exploitation and vector operations**

One of the advantageous side effects of the methods presented in this thesis concerning structure exploitation and vector-level differentiation is to make the reverse mode more practical in terms of space requirements. These methods are presented in Chapters 3 and 4.

Ideally we would like the reverse mode to have the same spatial and temporal complexity as the forward mode, in other words the temporal and spatial complexity should be only a constant factor of the function evaluation itself, thus making the reverse mode feasible, i.e.

$$\frac{\omega(f, \nabla f)}{\omega(f)} = O(1), \quad \frac{S(f, \nabla f)}{S(f)} = O(1).$$

## 1.3 Large Scale Optimization

Most optimization algorithms involve an iterative process to improve the current guess, say point $x_k$, by taking a step $s_k$ in a direction that reduces the objective function.

### Nonlinear equations example

For the nonlinear equations problem (finding a solution of $F(x) = 0$, where $F \in \Re^n \rightarrow \Re^n, x \in \Re^n$), this computation can be summarized as:

For $k = 1, 2, \ldots$

1. Compute step $s_k$.
2. $x_{k+1} = x_k + s_k$.

The first step of the algorithm, which computes the step $s_k$, typically involves computing or approximating derivative information of $F(x_k)$. A very special case is when $s_k$ is the Newton step, i.e. $s_k = -J(x_k)^{-1}F(x_k)$, where $J(x_k)$ denotes the $n \times n$ Jacobian matrix of $F$ at the current point $x_k$. The Newton step can be generalized for least-squares problems where $F \in \Re^n \to \Re^m, m \geq n$, and the Newton step is the least-squares approximate solution of $J(x_k)s_k = -F(x_k)$.

## Minimization example

In the minimization case, we deal with a scalar function $f(x)$ where $f \in \Re^n \to \Re$, and we are looking for a local minimum of $f(x)$. The Newton step algorithm to compute the minimizer can be written as:

For $k = 1, 2, \ldots$

1. Compute $s_k$ satisfying $\nabla^2 f(x_k)s_k = -\nabla f(x_k)$.
2. $x_{k+1} = x_k + s_k$.

One popular way to compute the Newton step $s = J^{-1}F$ is to compute or approximate the Jacobian matrix and then solve the linear system. The Jacobian matrix can be computed in $n$ functions evaluations by numerical finite differences: the $i$th column can be approximated by the difference $J(:, i) \approx \frac{F(x_k+\alpha e_i)-F(x_k)}{\alpha}$, hence requiring $n$ differences or $n$ extra function evaluations to compute the Jacobian matrix. We can also compute the Jacobian matrix using AD in forward mode by letting $V = I$, and the cost equation tells us that this also costs proportional to $n$ function evaluations. If $m < n$, then it might be cheaper to compute the Jacobian matrix using the reverse mode of automatic differentiation, by letting $W = I$, and the cost of computing the Jacobian matrix this way will be proportional to $m$ function evaluations.

Hence, we require a minimum of $\min(m, n)$ function evaluations to compute the Jacobian matrix without exploiting any sparsity or problem structure. This factor is unacceptable for many large scale problems where we are dealing with $m, n$ of the order of 100,000 or more. If the function evaluation takes a second, the Jacobian matrix evaluation for $m = n = 100,000$ will take $100,000$ seconds or about 28 hours!! Fortunately, most large scale problems exhibit sparsity (in derivative matrices) or some sort of problem structure which can be exploited for efficiency, and the number of function evaluations needed to compute the Jacobian matrix can often be brought down to a handful.

In the next section we present a trivial example of a structured problem to illustrate the kind of gains we can get by exploiting the problem structure. Andreas Griewank's tutorial on computational differentiation and optimization [Gri94] gives more information about the interplay between AD and optimization.

## 1.4 Exploiting Problem Structure

Let us motivate the need to exploit the problem structure by consider the following very simple composite function example. Assume that the nonlinear equations function $F$ has the form

$$F(x) = f_1(f_2(x)), \qquad \begin{array}{c} F \in \Re^n \rightarrow \Re^n \\ f_1 \in \Re^1 \rightarrow \Re^n, f_2 \in \Re^n \rightarrow \Re^1 \end{array}$$

That is, $f_1$ is a function mapping a scalar to a vector and $f_2$ maps a vector to a scalar. $f_1$ and $f_2$ can be any general nonlinear functions. The Jacobian of this function will look like

$$J = uv^t$$

where $u \in \Re^{n\times 1}$ is the Jacobian matrix of $f_1$, and $v^t$ is the Jacobian matrix of $f_2$. Now, since $J$ is a rank-1 product it will be completely dense and its computation will require work proportional to $n$ times the work to compute $F$. But, if we exploit structure by computing the components of the Jacobian matrix separately then $J$ can be computed in time proportional to one evaluation of $F$. $u$ can be evaluated by doing a forward pass on $f_1$, and $w$ can be computed by a reverse pass on $f_2$, which will take time proportional to one evaluation of $f_1$ plus one evaluation of $f_2$; hence the total time will be proportional to one evaluation of $F$. Here is the recipe to compute the Jacobian $J$:

1. Compute $v = \nabla f_2(x)$ via reverse mode.
2. Compute $u = \frac{df_1(y)}{dy}$ using forward mode.
3. Form $J = uv^T$.

This is of course just a toy example to illustrate what we mean by structure in the problem. In later chapters we develop the full framework of structure exploitation technology and present non-trivial applications.

## 1.5 Outline

This thesis proffers the use of automatic differentiation technology for optimization, illustrating the efficiency advantage. AD can provide benefits in a variety of ways: a) the raw derivative computation is fast, accurate and automatic or b) in exploiting problem structure or sparsity or c) in designing robust software to do numerical optimization and finally d) educating the user about the problem structure via this technology.

In Chapter 2, we present an efficient method to compute sparse Jacobian matrices of general nonlinear maps, by combining the forward and reverse modes of AD. This method, named "bi-coloring", attempts to reduce the computational cost involved to recover sparse Jacobians from thin matrix vector product pair $(W^T J, JV)$, choosing the matrices $W$ and $V$ as thin (with as few columns) as possible.

The major theme of this thesis, the problem structure exploitation methodology (EASE – **E**xtended functions **A**nd **S**tructure **E**xploitation) is presented in Chapter 3. Along with the description of the problem structure exploitation scheme, we also provide a general framework for classifying the problem structure itself and understanding various different structures via examples.

In Chapter 4 we stress the importance of using AD at the level of matrix-vector operations and describe the implementation of ADMAT, the MATLAB AD tool. In Chapter 5 we introduce a very important concept of semi-automatic differentiation where we present code templates for "surgical" application of AD tools to specific code segments in the program.

In Chapter 6, we present a number of applications where we employ EASE to dramatically reduce the computational cost. The applications range from engineering design problems to financial pricing problems to wave propagation reflection seismology problems to optimal control and dynamical systems problems.

In Chapter 7 we introduce the concept of structured algorithms for optimization, where we illustrate how optimization algorithm designers can benefit by "thinking structure". In Chapter 8 we summarize some ideas that lead to efficient parallel implementation of automatic differentiation technology. We summarize and conclude in Chapter 9 by presenting some future lines of work and mentioning some related work in AD technology relevant to work presented in this thesis. The use of AD as a derivative computing engine naturally automates all the methodologies presented in the work presented in this thesis . We present ways to make the numerical optimization software design very transparent. The user is just required to provide the function to be optimized. In the appendix we describe the ADMIT and ADMAT software briefly.

# Chapter 2

# Exploiting Sparsity in Derivative Matrices

The efficient numerical solution of nonlinear systems of algebraic equations, $F(x) : \Re^n \to \Re^m$, 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 $x$. The Bi-coloring approach we present here is a new efficient approach for minimizing the cost of computing a sparse Jacobian matrix of a nonlinear map, employing AD in the process. For a complete reference on bi-coloring refer to Coleman and Verma [CV98c].

Similarly efficient solution of a numerical optimization problem, $\min_x f(x)$ requires repeated calculation of the matrix of second derivatives, the Hessian matrix, $H(x) \in \Re^n \to \Re^n$. We review the previous work done on computing sparse Hessian matrices.

## 2.1 Computing Sparse Jacobian Matrices

**Sparse Jacobian problem:** 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 motivation for solving above problem comes from the following two observations on the problem of computing a sparse Jacobian matrix. Sparse finite-differencing literature [CPR74, CGM84, CM84a, CM84b, CGM85, CC86] provides a solution based on partitioning of columns, to define a matrix $V$ such that $J$ can be determined from the product $JV$. However, the matrix $V$ is not guaranteed to be thin, even if $J$ has a lot of sparsity: 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: consider the case where $J$ has a single dense column.

The bi-coloring method gets around this problem, and is provably better than 1-sided coloring always. Here is a simple example which demonstrates the "win" of bi-coloring. Consider the following $n$-by-$n$ Jacobian, symmetric in structure but not in value:

$$J = \begin{pmatrix} \square & \triangle & \triangle & \triangle & \triangle \\ \square & \diamond & & & \\ \square & & \diamond & & \\ \square & & & \diamond & \\ \square & & & & \diamond \end{pmatrix}. \tag{2.1}$$

It is clear that a partition of columns consistent with the direct determination of $J$ requires $n$ groups due to the presence of a dense row. Therefore, if the 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, and the reverse mode of AD will compute $J$ in time 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 $V = (e_1, e_2 + e_3 + e_4 + e_5)$; $W = (e_1)$. Clearly elements $\square$, $\diamond$ are directly determined from the product $JV$; elements $\triangle$ are directly determined from the product $W^T J$.

### 2.1.1 Direct and substitution methods

If we relax the restriction that each nonzero element of $J$ be determined directly then it is possible that the work required to evaluate the nonzeroes of $J$ can be further reduced. For example we could allow for a "substitution" process when recovering the nonzeroes of $J$ from the pair $(W^T J, JV)$. Figures 2.1 and 2.2 illustrate that a substitution method can win over direct determination: Figure 2.1 corresponds to direct determination, while Figure 2.2 corresponds to a determination using substitution.

$$\begin{pmatrix} \square_{11} & \triangle_{12} & \triangle_{13} & & & & & & \\ \square_{21} & & & & & & & & \\ \square_{31} & & & & & & & & \\ \square_{41} & & & \square_{44} & \triangle_{45} & \triangle_{46} & & & \\ & & & \square_{54} & & & & & \\ & & & \square_{64} & & & & & \\ & & & \square_{74} & & & \square_{77} & \triangle_{78} & \triangle_{79} \\ & & & & & & \square_{87} & & \\ & & & & & & \square_{97} & & \\ & & & & & & \square_{10,7} & & \end{pmatrix}$$

Figure 2.1: Optimal partition for direct method

In both cases elements labeled $\square$ are computed from the column grouping, i.e., calculated using the product $JV$; elements labeled $\triangle$ are calculated from the row groupings, i.e., calculated using the product $W^T J$. The matrix in Figure 2.1 indicates that we can choose $V = (e_1 + e_7, e_4)$ and $W = (e_1 + e_4 + e_7)$ and deter-

$$
\begin{pmatrix}
\square_{11} & \triangle_{12} & \triangle_{13} & & & & & & \\
\square_{21} & & & & & & & & \\
\square_{31} & & & & & & & & \\
\square_{41} & & & \triangle_{44} & \triangle_{45} & \triangle_{46} & & & \\
 & & & \square_{54} & & & & & \\
 & & & \square_{64} & & & & & \\
 & & & \square_{74} & & & \triangle_{77} & \triangle_{78} & \triangle_{79} \\
 & & & & & & \square_{87} & & \\
 & & & & & & \square_{97} & & \\
 & & & & & & \square_{10,7} & &
\end{pmatrix}
$$

Figure 2.2: Optimal partition for substitution method

mine all elements directly. Therefore in this case the work to compute $J$ satisfies $\omega(J) \sim 3 \cdot \omega(F)$. Note that some elements can be determined twice, e.g., $J_{11}$.

However, the matrix in Figure 2.2 shows how to obtain the nonzeroes of $J$, using substitution, in work proportional to $2 \cdot \omega(F)$. Let $V = W = (e_1 + e_4 + e_7)$. Then elements of $J$ can be resolved from $(JV, W^T J)$.

We can now state our main problem) more precisely.

> *The direct (*substituion*) bi-partition problem*: Given a matrix $A$, obtain the matrices $(V, W)$ which allow direct determination (*determination by substitution*) of elements in $A$ from the pair $(AV, W^T A)$ , such that the sum of number of columns in $V$ and $W$ is minimized.

The bi-partition problems can also be expressed in terms of graphs and graph coloring. This graph view is important in that it more readily exposes the relationship of the bi-partition problems with the combinatorial approaches used in the sparse finite-differencing literature, e.g., [CPR74,CGM84,CM84a,CM84b,CGM85, CC86].

### 2.1.2 Algorithms for direct and substitution bi-coloring

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 2.3. 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, $G_C$ or a set of groups of columns where each column belongs to one and only one group. $G_C$ defines the matrix $V$: every group of columns in the partition defines a column in $V$, with ones corresponding to the position of the columns and the rest as zero. The matrix $W$ is defined similarly by a partition of a subset of rows, $G_R$, which is given by a coloring of $\mathcal{G}_R^I$. We let $J_C$ denote both a set of pairs $(i, j)$ of positions in the matrix $J$ and the corresponding matrix, and similarly for $J_R$. No confusion should result.

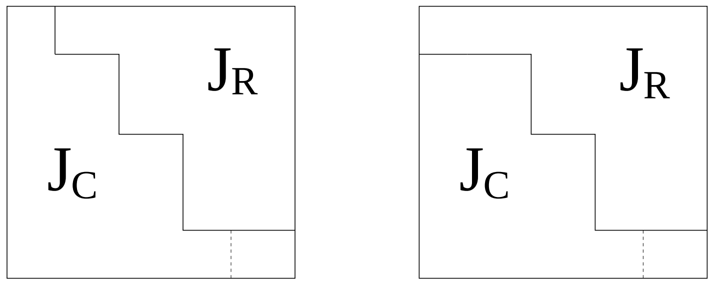



Figure 2.3: Possible partitions of the matrix $\tilde{J} = P \cdot J \cdot Q$

**Direct determination**

In the direct case the intersection graph $\mathcal{G}_C^I$ is defined: $\mathcal{G}_C^I = (\mathcal{V}_C^I, \mathcal{E}_C^I)$ where

- Vertex $j \in \mathcal{V}_C^I$ if $nnz(J_C e_j) \neq 0$.
- $(r, s) \in \mathcal{E}_C^I$ if $r \in \mathcal{V}_C^I, s \in \mathcal{V}_C^I$, $\exists k$ such that $J_{kr} \neq 0, J_{ks} \neq 0$ and **either** $(k, r) \in J_C$ **or** $(k, s) \in J_C$.

The key point in the construction of graph $\mathcal{G}_C^I$, and why $\mathcal{G}_C^I$ is distinguished from the usual column intersection graph, is that columns $r$ and $s$ are said to intersect if and only if their nonzero locations partially overlap in $J_C$: i.e., columns $r$ and $s$ intersect if $J_{kr} \cdot J_{ks} \neq 0$ and **either** $(k, r) \in J_C$ **or** $(k, s) \in J_C$ for some $k$.

The "transpose" of the procedure above is used to define $\mathcal{G}_R^I = (\mathcal{V}_R^I, \mathcal{E}_R^I)$. Specifically, $\mathcal{G}_R^I = (\mathcal{V}_R^I, \mathcal{E}_R^I)$ where

- Vertex $i \in \mathcal{V}_R^I$ if $nnz(J_R^T e_i) \neq 0$.
- $(r, s) \in \mathcal{E}_R^I$ if $r \in \mathcal{V}_R^I, s \in \mathcal{V}_R^I$, $\exists k$ such that $J_{rk} \neq 0, J_{sk} \neq 0$ and **either** $(r, k) \in J_R$ **or** $(s, k) \in J_R$.

The graph $\mathcal{G}_C^I(\mathcal{G}_R^I)$ is distinguished from the usual column(row) intersection graph in that two columns (rows) $r$ and $s$ are said to intersect if and only if their nonzero locations partially overlap in $J_C(J_R)$. The bi-partition $(G_R, G_C)$, induced by coloring of graphs $\mathcal{G}_R^I$ and $\mathcal{G}_C^I$, is consistent with direct determination of $J$. To see this refer to the bi-coloring paper [CV98c].

**Example:** Consider the example Jacobian matrix structure shown in Figure 2.4 with the partition $(J_C, J_R)$ shown.

The graphs $\mathcal{G}_C^I$ and $\mathcal{G}_R^I$ formed by the algorithm outlined above are given in Figure 2.5.

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

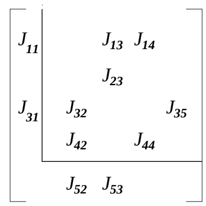



Figure 2.4: Example Partition

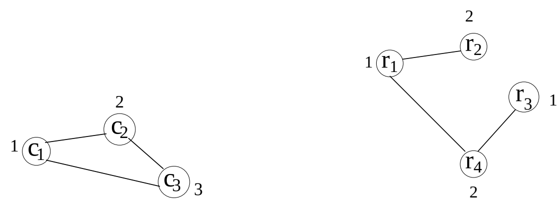



Figure 2.5: Graphs $\mathcal{G}_C^I$ and $\mathcal{G}_R^I$ (direct approach)

in that group:

$$V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad JV = \begin{pmatrix} J_{11} & 0 & \times \\ 0 & 0 & \times \\ J_{31} & \times & 0 \\ 0 & \times & 0 \\ 0 & J_{52} & J_{53} \end{pmatrix} \quad W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix} \quad J^T W = \begin{pmatrix} \times & \times \\ J_{32} & J_{42} \\ J_{13} & J_{23} \\ J_{14} & J_{44} \\ J_{35} & 0 \end{pmatrix}.$$

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

### 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.

In the substitution case the intersection graph $\mathcal{G}_C^I$ is defined as $\mathcal{G}_C^I = (\mathcal{V}_C^I, \mathcal{E}_C^I)$ where

- Vertex $j \in \mathcal{V}_C^I$ if $nnz(J_C e_j) \neq 0$.
- $(r, s) \in \mathcal{E}_C^I$ if $r \in \mathcal{V}_C^I, s \in \mathcal{V}_C^I$, $\exists k$ such that $J_{kr} \neq 0, J_{ks} \neq 0$ and **both** $(k, r) \in J_C$, $(k, s) \in J_C$.

The "transpose" of the procedure above is used to define $\mathcal{G}_R^I = (\mathcal{V}_R^I, \mathcal{E}_R^I)$. Specifically, $\mathcal{G}_R^I = (\mathcal{V}_R^I, \mathcal{E}_R^I)$ where

- Vertex $i \in \mathcal{V}_R^I$ if row $i \in J_R$ and $nnz(J_R^T e_i) \neq 0$.
- $(r,s) \in \mathcal{E}_R^I$ if $r \in \mathcal{V}_R^I, s \in \mathcal{V}_R^I$, $\exists k$ such that $J_{rk} \neq 0, J_{sk} \neq 0$ and **both** $(r,k) \in J_R$, $(s,k) \in J_R$.

The intersection graph $\mathcal{G}_C^I(\mathcal{G}_R^I)$ for substitution require complete overlap in $J_C(J_R)$. 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 2.6.

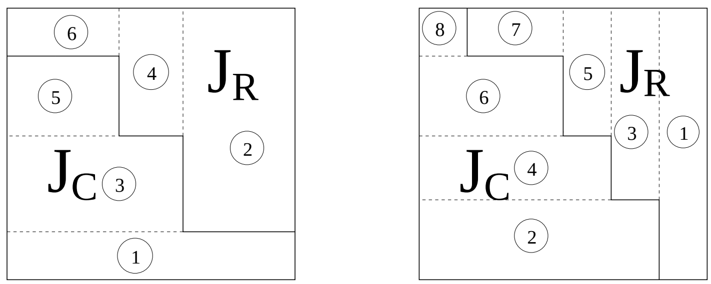



Figure 2.6: Substitution Orderings

Figure 2.6 illustrates two of four possible nontrivial types of partitions. In both cases it is clear that the 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 section “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 section “2” can be determined after “1” and so on.

**Algorithm for partitioning $J$.**

We now consider the problem of obtaining a useful partition $[J_C|J_R]$. Algorithm **MNCO** builds partition $J_C$ from the 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 shown in equation (2.2). Let $r$ be the best choice of a row to be added to $J_C$ and $c$ the best column under consideration to be added to $J_R$. We choose to add row $r$ to $J_C$ if equation (2.2) is satisfied else we add column $c$ to $J_R$. This lower bound effect is given by:

$$\rho(J_R^T) + \max(\rho(J_C), nnz(M^T e_r)) < \rho(J_C) + \max(\rho(J_R^T), nnz(M e_c)). \qquad (2.2)$$

where $\rho(A)$ is the maximum number of nonzeroes in any row of matrix $A$ and $M$ is the current (unassigned) submatrix of $J$.

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 $\rho(J_R^T)$, hence the LHS of equation

(2.2) is a lower bound on the number of colors required if $r$ is added, and the RHS is a lower bound on the number of colors required if $c$ is added and we make the choice which result in the smallest possible lower bound.

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 $R = (1 : m)$, $C = (1 : n)$, $M = J(R, C)$, $J_C = []$, $J_R = []$
2. Repeat Until $M = \emptyset$
   - Find $r \in R$ with fewest nonzeroes in $M^T e_r$
   - Find $c \in C$ with fewest nonzeroes in $Me_c$
   - if $\rho(J_R^T) + \max(\rho(J_C), nnz(M^T e_r)) < \rho(J_C) + \max(\rho(J_R^T), nnz(Me_c))$ (LB)

     $J_C$=$J_C \cup (r, j, j \in C, M(r, j) \neq 0)$

     $R$=$R$-$\{r\}$

     else

     $J_R$=$J_R \cup (i, c, i \in R, M(i, c) \neq 0)$

     $C$=$C$-$\{c\}$

     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**.

### 2.1.3 Performance of bi-coloring

Here is the performance of bi-coloring on a linear programming testbed (http://www.netlib.org/lp/data/) with results summarized in Table 2.1.

Table 2.1: Totals for LP Collection

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

The numbers represent the actual number of groups needed using the heuristic partition approach. Hence, these results must be taken with a pinch of salt due to the heuristic nature of the bi-coloring algorithm.

The substitution method is shown to give quite accurate answers, and is shown to be typically faster than the direct method. The experiments done using ADOL-C [GJU96] show that bi-coloring techniques always win over the finite-differencing technique, in terms of execution time.

### 2.1.4 Weighted bi-coloring

Implicit in our approach is the assumption that the cost of the forward mode is equal to that of the reverse mode. Typical AD tools have reverse mode twice as 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 $MNCO$ (§4.3) to heuristically solve a "weighted" problem, $\min w_1\chi_1 \; + \; w_2\chi_2$, where $\chi_1$ is the number of row groups (or colors assigned to the rows), and $\chi_2$ is the number of column groups (or colors assigned to the columns). The heuristic $MNCO$ can be changed to address this problem by simply changing the lower bound effect (equation (2.2)) to:

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

Different weights produce different allocations of work between forward and reverse modes, skewed to reflect the relative costs. For example, consider a 50-by-50 grid matrix with $DENS = 1$, (see Figures 2.1 and 2.2), and let us vary the relative weighting of forward versus reverse mode: $w_1 = 0 : .25 : 1$, and $w_2 = 1 - w_1$. The results of our weighted bi-coloring approach are given in Table 2.2.

Table 2.2: Weighted problem results

| $w_1$ | $\chi_1$ | $\chi_2$ |
|---|---|---|
| 0.00 | 50 | 0 |
| 0.25 | 16 | 7 |
| 0.50 | 10 | 10 |
| 0.75 | 7 | 16 |
| 1.00 | 0 | 50 |

## 2.2 Computing Sparse Hessian Matrices

Here we review the previous work for computing sparse Hessian matrices of nonlinear scalar maps, $f(x), x \in \Re^n$. The work on this subject was started by Powell and Toint [PT79], Coleman and Moré [CM84a] and Coleman and Cai [CC86] built on this work using graph-theoretic techniques to come up with efficient algorithms. In summary, the goal of computing a sparse symmetric Hessian matrix, $H(x) \in \Re^n \times \Re^n$, can be achieved by three different classes of methods.

1. **Ignoring the symmetry:** Given the sparsity pattern of Hessian, $SPH$, the method determines a permutation $p$ and a partition of the columns of $H$, consistent with the determination of all nonzeros of $H$ directly and independently. This class of method includes exactly the ones used for computing Jacobians since we are ignoring symmetry.

2. **Direct – exploiting symmetry:** Given the sparsity pattern of Hessian the method determines a permutation $p$ and a partition of the columns of $H$, consistent with the determination of all nonzeros of $H$ 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 [CM84a].

3. **Substitution – exploiting symmetry:**

   Given the sparsity pattern of Hessian the method determines a permutation $p$ and a partition of the columns of $H$, consistent with the determination of all nonzeros of $H$ 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 [CC86].

The following inequality holds:

$$\chi_i(H) \geq \chi_p(H) \geq \chi_c(H)$$

where $\chi_i(H), \chi_p(H), \chi_c(H)$ stand respectively for the chromatic numbers (or the number of groups needed) for the method which ignores symmetry altogether, the path chromatic number and the cyclic chromatic number. This inequality is easy to see because the partitioning scheme becomes less and less restrictive as you go from the method ignoring symmetry to the substitution method.

# Chapter 3

# Exploiting Structure

As we have seen, straightforward application of $AD$ software on large-scale problems can require an inordinate amount of computation. Fortunately, large-scale nonlinear problems typically exhibit either sparsity or structure in their Jacobian or Hessian matrices. In this chapter we proffer general approaches for exploiting structure to yield efficient ways to determine Jacobian and Hessian matrices (and Newton steps) via automatic differentiation. The technique demonstrated in this chapter is referred to as EASE, which stands for **E**xtended functions **A**nd **S**tructure **E**xploitation.

## 3.1 Structure

A fundamental computation with regard to a nonlinear system, $F : \Re^n \rightarrow \Re^m$, is the evaluation of the Jacobian matrix $J$ of $F$ at any given argument $x$: $J(x) \in \Re^{m \times n}$; in the optimization concerning a scalar function $f \in \Re^n \rightarrow \Re$ computing the Hessian matrix $H$ is the main task.

For large problems the computation of $J$ or $H$ by a straightforward application of either mode of $AD$ can be unacceptably expensive. Recently, techniques for the efficient determination of *sparse* Jacobian matrices $J$, via $AD$, have been developed [AMB+94,CV98c,SH95],e.g. the bi-coloring approach of Coleman and Verma [CV98c], as discussed in Chapter 2. Unfortunately, not all large systems exhibit sparse Jacobian or Hessian matrices. In this section, we demonstrate the exploitation of structure in large-scale applications and show how it is possible to dramatically lower the cost of computing $J$ or $H$ by exploiting structure in the user computation.

### 3.1.1 Structure in Jacobian matrices

The following composite structure is common in large-scale problems:

$$F(x) = \bar{F}(y) \tag{3.1}$$

where $y$ is the solution to a large sparse positive definite system,

$$Ay = \tilde{F}(x), \tag{3.2}$$

and $A = A(x)$. The Jacobian of $F(x)$, $J(x)$, is almost always dense even when matrices $\bar{J}, \tilde{J}$, and $A_x y$ are sparse (which is typical) where $\bar{J}$ is the Jacobian of $\bar{F}$ with respect to $y$, $\tilde{J}$ is the Jacobian of $\tilde{F}$, and $A_x y$ is the Jacobian of the mapping $A(x)y$ (with respect to $x$). To see this consider that

$$J = \bar{J}A^{-1}[\tilde{J} - A_x y]. \tag{3.3}$$

It is the application of $A^{-1}$ that causes matrix $J$ to be dense – this will almost surely be the case unless $A^{-1}$ is very special, e.g., diagonal.

So, direct application of *sparse AD* techniques offers no advantage in this case. However, it is possible to exploit the structure of this composite function and apply the sparse *AD* techniques at a deeper level. To see this consider the following "program" to evaluate $z = F(x)$, given $x$:

| | | |
|---|---|---|
| *"Solve" for* $y_1$ : | $y_1 - \tilde{F}(x) = 0$ |
| *Solve for* $y_2$ : | $Ay_2 - y_1 = 0$ |
| *"Solve" for* $z$ : | $z - \bar{F}(y_2) = 0.$ |

But this program can be viewed as a nonlinear system of equations in $(x, y_1, y_2)$ with corresponding Newton equations:

$$J_E \begin{pmatrix} \delta_x \\ \delta_{y_1} \\ \delta_{y_2} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ F(x) \end{pmatrix}, \tag{3.4}$$

where

$$J_E = \begin{bmatrix} -\tilde{J} & I & 0 \\ A_x y_2 & -I & A \\ 0 & 0 & -\bar{J} \end{bmatrix}. \tag{3.5}$$

Here is a key point: the "extended" Jacobian matrix $J_E$ is sparse and clearly sparse AD-techniques, e.g., [CV98c], can be applied with respect to

$$F_E(x, y) = \begin{pmatrix} y_1 - \tilde{F}(x) \\ A(x)y_2 - y_1 \\ -\bar{F}(y_2) \end{pmatrix}. \tag{3.6}$$

to efficiently determine $J_E$. For example, the work required by the bi-coloring technique developed in [CV98c] is $\chi \cdot \omega(F_E) = \chi \cdot \omega(F)$ where $\chi$ is a "bi-chromatic number" dependent on the sparsity of $J_E$. Typically, $\chi << \min(m, n)$. Additional linear algebra work is needed to extract $J$ from $J_E$: compute the Schur complement (introducing zero matrices in positions $(3,2), (3,3)$) and obtain,

$$J = \bar{J}A^{-1}[\tilde{J} - A_x y].$$

If the Newton step $\delta_x = -J^{-1}F(x)$ is required, then it is not necessary to explicitly form $J$. For example, the extended system (3.4) can be solved directly.

This can afford significant savings. To illustrate, consider the following experiment. We define a composite function $F(x)$ following the form described above.

The functions $\tilde{F}$ and $\bar{F}$ are defined to be the Broyden [Bro65] function (the Jacobian is tridiagonal). The structure of $A$ is based on the 5-point Laplacian defined on a regular $\sqrt{n}$-by-$\sqrt{n}$ grid. Each nonzero element of $A(x)$ depends on $x$ in a trivial way such that the stucture of matrix $A_x \cdot v$, for an arbitrary vector $v$, is equal to the structure of matrix $A$. In particular, for all $(i, j)$, $i \neq j$ where $A_{ij}$ is nonzero the function $A_{ij}(x)$ is defined, $A_{ij} = x_j$.

Figure 3.1 plots the time to calculate the Newton step, given $J_E$, via the formation of $J$ using (3.3) versus the computation of the Newton step using a direct sparse solve for equation (3.4). Experiments were perfomed in MATLAB, with sparse system solving implemented using the "backslash" function. All matrices are sparse in this example except for the final Jacobian matrix $J$. Clearly it pays to avoid the formation of $J$ and the advantage grows with $n$.

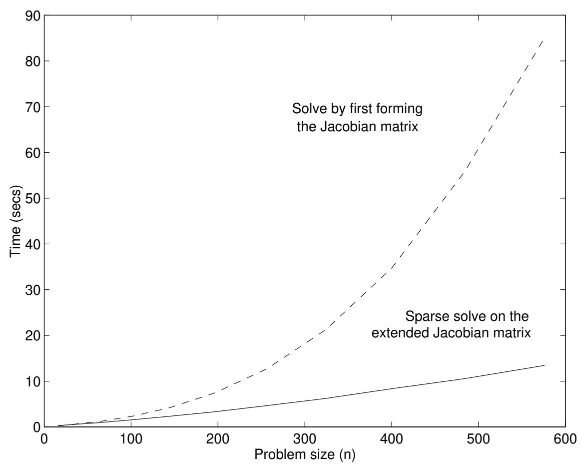



Figure 3.1: Comparison of two approaches to calculate the Newton step

It is also possible to compute an approximate Newton step, without forming $J$, using an iterative solver. Specifically, if a sparse factorization of $A$ is computed, an iterative solver involving only matrix-vector products can be applied to

$$(\bar{J}A^{-1}[\tilde{J} - A_x y])s \;=\; -F(x). \tag{3.7}$$

In this chapter we illustrate how these ideas can be applied more generally: in many cases the natural "coarse-grained" program yields a sparse "extended" Jacobian matrix which in turn, can be efficiently computed by sparse AD-techniques.

### Generalising the structure

Large-scale nonlinear systems $F(x) = 0$ often exhibit a natural lower Hessenberg form as shown in figure 3.2. Usually, an easy programmatic way to describe $F$, at a high level, is to state this lower Hessenberg description, or program, $F_E$. The corresponding Jacobian of $F_E$, $J_E$, is typically a sparse matrix: sparse $AD$ techniques can be applied to efficiently compute $J_E$. The Jacobian of $F$ and/or the Newton step $\delta_x = -J^{-1}F(x)$ can subsequently be computed from $J_E$. Two key points are:

- The matrix $J_E$, though larger than $J$, is usually considerably sparser.
- The high level program $F_E$ is usually readily available to the user: it is often the most natural way of expressing $F$.

To be more precise, a natural way to evaluate the nonlinear systems $z = F(x)$ is via the lower Hessenberg program illustrated in Figure 3.2 where we assume equation $i$ *uniquely* determines $y_i$, $i = 1 : p$. We take the point of view that the function $F_E$

$$
\boxed{
\begin{array}{ll}
\textit{Solve for } y_1: & F_1(x, y_1) = 0 \\
\textit{Solve for } y_2: & F_2(x, y_1, y_2) = 0 \\
\quad\vdots & \\
\textit{Solve for } y_p: & F_p(x, y_1, y_2, \ldots, y_p) = 0 \\
\textit{``Solve'' for output } z: & -z + F_{p+1}(x, y_1, y_2, \ldots, y_p) = 0
\end{array}
}
$$

Figure 3.2: A General Structured Computation

is explicitly available, where

$$F_E = \begin{pmatrix} F_1 \\ F_2 \\ \vdots \\ F_p \end{pmatrix}.$$

Indeed, usually the component functions of $F_E$, $F_i$, $i = 1 : p$, are conveniently available to the user. Many well-known structured problems are covered by this view: e.g., partially-separable functions, dynamical (recursive) systems, composite functions, various gradient computations.

The program in Figure 3.2 can be differentiated to give the extended Newton system:

$$J_E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -F \end{pmatrix} \tag{3.8}$$

where

$$J_E = \left(\begin{array}{c|cccc}
\frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\
\frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\
\vdots & \vdots & & \ddots & \\
\frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \cdots & & \frac{\partial F_p}{\partial y_p} \\
\hline
\frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial y_1} & \cdots & & \frac{\partial F_{p+1}}{\partial y_p}
\end{array}\right). \tag{3.9}$$

If each "intermediate vector" $y_i$ is a unique and differentiable function of its arguments $(x, y_i, \ldots y_{i-1})$ then the super-diagonal of (3.9) has nonsingular blocks, $\frac{\partial F_i}{\partial y_i}$, and $\delta_x$ is the Newton step $\delta_x = -J^{-1}F(x)$. A result connecting nonsingularity of the extended Jacobian and the original Jacobian matrix is proved in Theorem 3.1.1. Note that we need both the uniqueness and differentiabililty conditions, e.g., the cuberoot function defining $y_i(x)$ as $F_1(x, y_1) = y_1^3 - x = 0$ defines $y_1$ uniquely but is nondifferentiable at $x = 0$; this function results in a singular $\frac{\partial F_1}{\partial y_1}$ at the argument $x = 0$.

We contend that $J_E$ will likely be sparse, considerably sparser than $J$; hence, sparse $AD$ techniques can be used to obtain $J_E$. The Jacobian matrix $J$ can be computed from $J_E$ by zeroing the $(2, 2)$-block row in $J_E$ using block Gauss transformations. Matrix $J$ shows up in the $(2, 1)$ location after elimination of the $(2, 2)$-block row.

If it is the Newton step that is required, and not matrix $J$ per se, then there are two natural alternatives to the explicit computation of $J$, given $J_E$. First, system (3.8) can be solved directly using a direct sparse factorization[1]: this is clearly an attractive option in some cases, e.g., Figure 3.1. A second possibility is to perform the $(2, 1)$-elimination symbolically to produce a "product form" expression for $J$, in the $(2, 1)$ location, which could then be used in any iterative linear solver requiring only matrix-vector products.

### Gradient computation

An important special case of Jacobian evaluation is the computation of the gradient of a scalar valued function, $f : \Re^n \to \Re$. The gradient of $f$ is merely the transpose of the Jacobian of $f$; hence, the last "block row" of $J_E$ in (3.9) is a single row vector. In general the strategies discussed above cannot improve upon a direct reverse-mode application of $AD$ to evaluate the gradient, in terms of time, since the reverse mode computes $\nabla f(x)$ in time proportional to $\omega(f)$. However, unveiling underlying stucture as discussed above can certainly help significantly if only forward-mode $AD$ is to be used.

For example, consider the case where $f$ is a partially separable function, $f = f_1 + f_2 + \cdots + f_p$, where $f_i : \Re^n \to \Re$, $i = 1 : p$, and each component function $f_i$ depends on only a few components of $x$. A natural way to evaluate $f$ at a given argument $x$ is to evaluate

$$F(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_p(x) \end{pmatrix}$$

and then sum the components of $F(x)$. A program to do this can be written:

[1] Griewank [Gri90] has proposed a similar idea in a more extreme form: $F_E$ is defined with respect to *all* intermediate variables. The resulting extended Jacobian matrix $J_E$ is huge, but very sparse.

$$\boxed{\begin{array}{lrcl} \textit{“Solve” for } y\text{:} & F(x) - y & = & 0 \\ \textit{Output } z: & -z + e^T y & = & 0 \end{array}}$$

In this case

$$J_E = \begin{pmatrix} J_F & -I \\ 0 & e^T \end{pmatrix}. \tag{3.10}$$

Clearly $J_E$ can be computed in time proportional to $\chi_I(J_F) \cdot \omega(f)$ by applying the bi-coloring/$AD$ approach to $F$ to obtain $J_F$. This special case, the efficient determination of a gradient of a partially separable function via the forward-mode of $AD$, is studied in detail in [BBKM95].

### 3.1.2 Structure in Hessian matrices

Since the Hessian matrix is the Jacobian of the gradient, the structure exploitation in this context is related closely to structure exploitation for the Jacobian matrix. The first question we address in this section is how to write a structured program to evaluate the gradient of $f$, $\nabla_x f$, such that automatic differentiation can be applied to yield second-derivative information in an efficient way. Once we have sorted this out, we take a step backwards and consider the more practical concern: how do we apply automatic differentiation directly to the structured program that evaluates $f$ to yield the Hessian matrix, or perhaps the Newton step, in an efficient way.

Consider a compact program to evaluate a scalar function $f \in \Re^n \rightarrow \Re$ as given in Figure 3.3.

$$\boxed{\begin{array}{l} \textit{Solve for } y: \ \tilde{F}^E(x, y) = 0 \\ \textit{“Solve” for output } z: \ z = \bar{f}(x, y_1, y_2, \ldots, y_p) \end{array}}$$

Figure 3.3: Structured $f$-Evaluation in Compact Form

where

$$\tilde{F}^E = \begin{pmatrix} F^1 \\ F^2 \\ \vdots \\ F^p \end{pmatrix}.$$

The “extended” Jacobian matrix,

$$J_E = \begin{pmatrix} \tilde{F}^E_x & \tilde{F}^E_y \\ \nabla_x \bar{f}^T & \nabla_y \bar{f}^T \end{pmatrix}. \tag{3.11}$$

Typically the Jacobian matrix $\tilde{J}^E = (\tilde{F}^E_x, \tilde{F}^E_y)$ is sparse and so sparse AD techniques can be applied to function $\tilde{F}^E$ to obtain this derivative information efficiently. Note also that $\tilde{F}^E_y$ is block lower-triangular, and, due to the assumption that the intermediate vectors $y$ are uniquely determined through a differentiable mapping, $\tilde{F}_y^E$ is nonsingular.

### The structured computation of the gradient

To answer the first question, the gradient of the structured function $f$ can be evaluated as illustrated in Figure 3.4.

| |
|---|
| *1. Differentiate* $\tilde{F}^E$ *yielding* $\tilde{J}^E = (\tilde{F}_x^E, \tilde{F}_y^E)$ |
| *2. Solve* $(\tilde{F}_y^E)^T w = -\nabla_y \bar{f}$. |
| *3. Set* $\nabla_x f = \nabla_x \bar{f} + (\tilde{F}_x^E)^T \cdot w$. |

Figure 3.4: A Structured Gradient program

The derivation of this program is simple: First differentiate the extended function $F_E$ to obtain $J_E$; then, eliminate the $(2,2)$-block, $\nabla_y \bar{f}^T$, to define vector $w$. Finally, modify the $(2,1)$-block of matrix $J_E$ using $w$ to get $\nabla_x f$. In other words, form matrix $J_E$ (3.11) and then eliminate the $(2,2)$-block using a block Gaussian transformation.

Inspired by this simple program to evaluate the gradient of $f$, we define an "extended" gradient $GF_E$, a vector function of the triple $(x, y, w)$:

$$GF_E(x, y, w) = \begin{pmatrix} \tilde{F}^E \\ (\tilde{F}_y^E)^T w + \nabla_y \bar{f} \\ \nabla_x \bar{f} + (\tilde{F}_x^E) w \end{pmatrix}.$$

In principle the vector function $GF_E$ can be differentiated, with respect to $(x, y, w)$, to yield a Newton system,

$$H_E \begin{pmatrix} \delta x \\ \delta y \\ \delta w \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -\nabla_x f \end{pmatrix}$$

where

$$H_E = \begin{pmatrix} \tilde{F}_x^E & \tilde{F}_y^E & 0 \\ (\tilde{F}_{yx}^E)^T w + \nabla_{yx}^2 \bar{f} & (\tilde{F}_{yy}^E)^T w + \nabla_{yy}^2 \bar{f} & (\tilde{F}_y^E)^T \\ (\tilde{F}_{xx}^E)^T w + \nabla_{xx}^2 \bar{f} & (\tilde{F}_{xy}^E)^T w + \nabla_{xy}^2 \bar{f} & (\tilde{F}_x^E)^T \end{pmatrix},$$

where, for instance, $(\tilde{F}_{yx}^E)^T w$ the Jacobian matrix of the vector function $(\tilde{F}_x^E) w$ w.r.t. $y$.

Typically matrix $H_E$ exposes more sparsity than matrix $H$ – the composite function described in Section 1 is a good illustration. Moreover, additional sparsity gains can often be achieved, in principle, if the structure in Step 3. of the gradient-evaluation program is exploited. In particular, notice that the computation $\nabla_x f = \nabla_x \bar{f} + (\tilde{F}_x^E)^T \cdot w$ exhibits "partially separable" form. It is often worthwhile to further break down this step as illustrated in [CV96b].

The (reduced or original) Hessian matrix $H$ is available from $H_E$ through a simple block-elimination procedure. For, example if we partition $H_E$,

$$H_E = \left(\begin{array}{c|c} A & L \\ \hline B & M \end{array}\right) \tag{3.12}$$

then $H = B - ML^{-1}A$.

**How to differentiate $f$ (twice)**

If we define $g(x, y, w) = \bar{f} + w^T \tilde{F}^E(x, y)$ then $H_E$ can be written:

$$H_E = \begin{pmatrix} \tilde{F}_x^E & \tilde{F}_y^E & 0 \\ \nabla_{yx}^2 g & \nabla_{yy}^2 g & (\tilde{F}_y^E)^T \\ \nabla_{xx}^2 g & \nabla_{xy}^2 g & (\tilde{F}_x^E)^T \end{pmatrix}. \tag{3.13}$$

This is an important observation because it yields the answer to the second major question of Section 2: How do we apply automatic differentiation directly to the $f$-evaluation code to yield the extended Hessian $H_E$ in an efficient way? The recipe follows from (3.13) and the definition of $w$:

1. Using the sparse AD techniques developed in [CV96b] compute the extended Jacobian $(\tilde{F}_x^E, \tilde{F}_y^E)$

2. Solve the block lower triangular system for $w$: $(\tilde{F}_y^E)^T w + \nabla_y \bar{f} = 0$.

3. Using sparse AD techniques, twice differentiate $g(x, y, w) = \bar{f} + w^T F^E(x, y)$, with respect to $x$, $y$, to determine the Hessian matrix, i.e., $H_E$. As indicated in Section 2.1, it can be advantageous to exploit the partially separable structure in $g(x, y, w) = \bar{f} + \sum_{i=1}^k w_i^T F_i^E$: i.e., compute the Hessian matrix of each component function $w_i^T F_i^E$ in turn.

Symmetry in the extended form $H_E$ can be achieved with (block) permutations:

$$H_E^S = \begin{pmatrix} 0 & \tilde{F}_y^E & \tilde{F}_x^E \\ (\tilde{F}_y^E)^T & \nabla_{yy}^2 g & \nabla_{yx}^2 g \\ (\tilde{F}_x^E)^T & \nabla_{xy}^2 g & \nabla_{xx}^2 g \end{pmatrix}. \tag{3.14}$$

## A result about extended Jacobian and Hessians matrices

This important result applies to both extended Jacobian matrices (equation (3.8)) and extended Hessian matrices (equation (3.13)).

**Theorem 3.1.1** *If $B_3$ is nonsingular, then*

$$C = \left(\begin{array}{c|c} B_1 & B_3 \\ \hline B_2 & B_4 \end{array}\right)$$

*is singular if and only if $D = B_2 - B_4 B_3^{-1} B_1$ is singular.*

**Proof :**

- $D$ **is singular** $\Rightarrow$ $C$ **is singular.** Since $D$ is singular, $\exists s \neq 0$ s.t $Ds = 0$. Then it is easy to see that vector $v = (s, -B_3^{-1}B_1 s)$ which is nonzero because $s \neq 0$ satisfies $Cv = 0$.

- $C$ **is singular** $\Rightarrow$ $D$ **is singular.** Since $C$ is singular, $\exists v \equiv (s, \tilde{s}) \neq 0$ s.t. $Cv = 0$, expanding:

$$\begin{aligned} B_1 s + B_3 \tilde{s} &= 0 \\ B_2 s + B_4 \tilde{s} &= 0 \end{aligned}$$

  Note that if $s = 0$, $\tilde{s}$ satisfies $B_3\tilde{s} = 0$ implying $\tilde{s} = 0$, so $s \neq 0$. Substituting $-B_3^{-1}B_1 s$ for $\tilde{s}$ in the equation $B_2 s + B_4\tilde{s} = 0$, gets us $(B_2 - B_4 B_3^{-1} B_1)s = 0$, or in other words $Ds = 0$, where $s \neq 0$, hence $D$ is singular.

$\square$

As a corollary it follows that the Matrix $C$ is nonsingular iff the the derivative matrix $D \equiv B - M(L-I)^{-1}A$ is nonsingular. In case of $C$ being extended Jacobian matrix, $D$ is the Jacobian matrix and if $C$ is the extended Hessian matrix $D$ will denote the true Hessian matrix.

### 3.1.3 Examples of structured problems

We discuss three common classes of structured nonlinear systems. In each case the Jacobian matrix is potentially dense; however, by differentiating the natural high-level program to compute $F$ we get a considerably sparse extended Jacobian matrix. The result is often a dramatic increase in efficiency.

#### Composite functions and dynamical systems

A composite function $F : \Re^n \rightarrow \Re^m$ can be written as:

$$F(x) = \bar{F}(T_p(T_{p-1}(\ldots(T_1(x))\ldots), \tag{3.15}$$

where, in general, functions $\bar{F}, T_i, i = 1 : p$ are vector maps. Recursively applying the chain rule yields $J$, the Jacobian of $F(x)$:

$$J = \bar{J} \cdot J_p \cdot J_{p-1} \cdot \ldots \cdot J_1$$

where $J_i$ is the Jacobian of $T_i$ evaluated at $T_{i-1}(T_{i-2}(\ldots(T_1(x))\ldots)$; $\bar{J}$ is the Jacobian of $\bar{F}$ evaluated at argument $T_p(T_{p-1}(\ldots(T_1(x))\ldots)$.

A natural high-level program to evaluate $F$ is given below where we let $y_0$ denote $x$:

*for* $i = 1 : p$
*"Solve" for* $y_i$*:* $y_i - T_i(y_{i-1}) \;=\; 0$
*"Solve" for* $z$*:* $z - \bar{F}(y_p) \;=\; 0.$

Clearly this program is a special case of the general form $F_E$ given in Figure 3.2, with corresponding Jacobian matrix $J_E$:

$$J_E = \begin{pmatrix} -J_1 & I & & & \\ & -J_2 & I & & \\ & & \ddots & \ddots & \\ & & & -J_p & I \\ & & & & -\bar{J} \end{pmatrix}$$

For example, consider the special case of a dynamical system: $T_i = T, \;\; i = 1 : p-1$, $\bar{F} = T$, $T : \Re^n \to \Re^n$ is a square nonsingular mapping, and

$$F(x) = T(T(\ldots(T(x))\ldots).$$

Typically the mapping $T$ is a sparse mapping which has a sparse Jacobian matrix. Assume that the Jacobian of $T$ is a tri-diagonal Jacobian matrix. For sufficiently large $p$ the Jacobian matrix $J$ will be dense; hence determination of $J$ by direct application of automatic differentiation requires $O(n \cdot \omega(F)) = O(n \cdot p \cdot \omega(T))$ flops. Therefore, the direct determination of $J$ followed by a direct solve requires $O(n \cdot p \cdot \omega(T) + n^3)$ flops. However, the determination of $J_E$ requires $O(p \cdot \omega(T))$ flops and solution of the extended system takes $O(n \cdot p)$ flops for a total of $O(p \cdot \omega(T) + n \cdot p)$ flops – generally, a much more attractive order of work.

### Generalized partial separability

We define a *generalized* partially separable vector-valued function,

$$F(x) = \bar{F}(y_1, y_2, \ldots, y_p); \qquad y_i = T_i(x), \quad i = 1, 2, \ldots, p. \tag{3.16}$$

Note that if function $\bar{F}$ is simply a summation then $F$ reduces to the usual notion of partial separability.

Following the general form given in Figure 3.2, $F$ can be computed with the following program,

| |
|---|
| *for* $i = 1 : p$ |
|     *"Solve" for* $y_i$*:* $y_i - T_i(x) = 0$ |
| *"Solve" for* $z : z - \bar{F}(y_1, y_2, \ldots, y_p) = 0$ |

Of course this program can be inefficient if some of the functions $T_i$ share common sub-expressions. Therefore a more general program can be written if we define a "stacked" vector $Y^T = (y_1^T, \ldots y_p^T)$ and a corresponding vector function

$$\tilde{F}(x) = \begin{pmatrix} T_1(x) \\ T_2(x) \\ \vdots \\ T_p(x) \end{pmatrix}.$$

This yields the simple 2-liner:

$$\boxed{\begin{array}{l} \text{“Solve” for } Y: \; Y - \tilde{F}(x) = 0 \\ \text{“Solve” for } z: \; z \; = \; \bar{F}(y_1, y_2, \ldots, y_p). \end{array}}$$

The general extended system (3.8) reduces to

$$\left(\begin{array}{c|ccccc} -\tilde{J}_1 & I & & & \\ -\tilde{J}_2 & & I & & \\ \vdots & & & \ddots & \\ -\tilde{J}_p & & & & I \\ \hline 0 & \bar{J}_1 & \bar{J}_2 & \cdots & \bar{J}_p \end{array}\right) \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ -F \end{pmatrix} \qquad (3.17)$$

It is clear that the computation of $J_E$ will, in general, be considerably more economical than than the straightforward application of $AD$ to determine
$J \;=\; \sum_{i=1:p} \tilde{J}_i \cdot \bar{J}_i$ via $AD$.

### A remark

Our two main examples, composite functions and generalized partially separable functions, are complementary in a structural sense. The evaluation of a composite function is a *depth* computation: each subsequent intermediate variable $y_i$ depends on the previous intermediate $y_{i-1}$. The computational graph of a composite function is a long chain: See Figure 3.5.

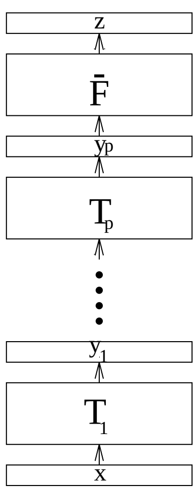



Figure 3.5: Composite Function

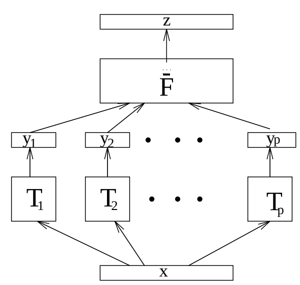



Figure 3.6: Generalized Partially Separable Function

In contrast, generalized partially separable functions are primarily breadth computations: typically, the intermediate variables $y_i$ are relatively independent of each other: it is the final computation $z = \bar{F}(y_1, y_2, \ldots, y_p)$ that binds the intermediates together. The computational graph is short and fat; see Figure 3.6.

Despite such grossly different structures the lower Hessenberg format, illustrated in Figure 3.2, is applicable in both cases: this view represents a convenient way to program the evaluation of $F$ and allows for the efficient application of $AD$ tools to determine the Jacobian matrix and/or the Newton step.

**Example of a structured Hessian problem**

In this section we illustrate the application of the structural ideas developed with an example from composite functions and dynamical systems.

A general composite function $f : \Re^n \rightarrow \Re^1$ can be written

$$z = f(x) = \bar{f}(T_p(T_{p-1}(\ldots(T_1(x))\ldots), \tag{3.18}$$

where, in general, functions $T_i, i = 1 : p$, are vector maps, while $\bar{f}$ is a scalar map. This formulation is very common, for example, in weather simulations. A natural high-level program to evaluate $F$ is given below, where we let $y_0$ denote $x$:

$$\boxed{\begin{array}{l} \textit{for } i = 1 : p \\ \textit{“Solve” for } y_i\textit{: } y_i - T_i(y_{i-1}) \;=\; 0 \\ \textit{“Solve” for } z\textit{: } z - \bar{f}(y_p) \;=\; 0. \end{array}}$$

Clearly this is a special case of the general extended form given in Figure 3.2, where we define

$$\tilde{F}^E = \begin{pmatrix} T_1(x) - y_1 \\ T_2(y_1) - y_2 \\ \vdots \\ T_k(y_{k-1}) - y_k \end{pmatrix}.$$

The general recipe given in Section 2.2 can be applied here, followed by a simple permutation, to yield the (symmetric) extended Hessian matrix $H_E^S$. If we define $g \;=\; \bar{f} + w^T \tilde{F}^E$ then

$$H_E^S = \begin{pmatrix} 0 & \tilde{F}_y^E & \tilde{F}_x^E \\ (\tilde{F}_y^E)^T & \nabla_{yy}^2 g & \nabla_{yx}^2 g \\ (\tilde{F}_x^E)^T & \nabla_{xy}^2 g & \nabla_{xx}^2 g \end{pmatrix}.$$

Additional structure can be gleaned by closer examination of the submatrices comprising $H_E^S$, i.e., exploiting the partial separable structure of $w^T F^{\tilde{E}}$:

$$H_{EE}^S = \left(\begin{array}{ccccccccc} 0 & \cdots & \cdots & 0 & -I & & & & J_1 \\ \vdots & & & \vdots & J_2 & -I & & & \\ \vdots & & & \vdots & & \ddots & \ddots & & \\ 0 & \cdots & \cdots & 0 & & & J_k & -I & \\ -I & (J_2)^T & & & \nabla_{y_1,y_1}^2 g & & & & \\ & -I & \ddots & & & \nabla_{y_2,y_2}^2 g & & & \\ & & \ddots & (J_k)^T & & & \ddots & & \\ & & & -I & & & & \nabla_{y_k,y_k}^2 & \\ (J_1)^T & & & & & & & & \nabla_{x,x}^2 g \end{array}\right)$$

Due to the high degree of sparsity, sparse AD techniques can be very effective in the direct determination of $H_{EE}^S$.

## 3.2 Stacked Jacobian and Hessian Matrices

Stacked Jacobian and Hessian matrices frequently arise in structured problems, especially partially separable problems. In this section we look at them carefully and prove some results about their computation.

### Stacked Jacobian matrices

There are basically two types of stacking that arise in Jacobian matrices, *vertical* as in partial separability, and *horizontal* which arises due to the possibility to split the arguments of a nonlinear function. We also see *diagonal* stacking as in the composite function example, which can be expressed as a combination of the vertical and horizontal stacking.

1. **Partial function separability**

   Consider a general partially separable nonlinear function $F : \Re^n \to \Re^m$ which has $p$ components, $F = F_1 + F_2 + \cdots + F_p$, where each of $F_i$ depend on only a small subset of values in $x$..

   Define $\tilde{F} : \Re^n \to \Re^{m \cdot p}$ as :

$$\tilde{F}(x) = \begin{pmatrix} F_1(x) \\ F_2(x) \\ \vdots \\ F_p(x) \end{pmatrix}.$$

   The Jacobian matrix for $\tilde{F}$ (note the connection to the extended Jacobian matrix) is:

$$\tilde{J} = \begin{pmatrix} J_1 \\ J_2 \\ \vdots \\ J_p \end{pmatrix}.$$

   **Result 3.2.1** *The number of groups required in a one-sided column coloring for the computation of sparse $\tilde{J}$ is less than the number of groups required by $J$.*

   To establish the above result consider the case with $p = 2$ and the following product

$$\tilde{J}V = \begin{pmatrix} J_1 V \\ J_2 V \end{pmatrix}$$

If $V$ is consistent with determination of $J$, then $V$ will also be consistent with determination of both $J_1$ and $J_2$ since the column intersection graphs of $J_1$ and $J_2$ will be subset of column intersection graph of $J$. $\Box$.

Note that the above result does not hold for one sided row coloring or for bi-coloring schemes, since the row intersection graph will be denser.

2. **Partial variable separability**

   Define $\hat{F} : \Re^{n \cdot p} \to \Re^m$

$$\hat{F}([x_1 x_2 \cdots x_p]) = \begin{pmatrix} F_1(x_1) & F_2(x_2) & \cdots & F_p(x_p) \end{pmatrix}$$

   i.e. give each constituent function a private copy of independent variables. The Jacobian matrix for $\hat{F}$ will look like:

$$\hat{J} = \begin{pmatrix} J_1 & J_2 & \cdots & J_p \end{pmatrix}$$

   **Result 3.2.2** *The number of groups required in one-sided column coloring for computation of sparse $\hat{J}$ are less than number of groups required by $J$.*

   Proof is the same as the one for Result 3.2.1.

3. **Combining partial separability and argument separability**

   Define $\bar{F} : \Re^{n \cdot p} \to \Re^{m \cdot p}$

$$\bar{F}([x_1 x_2 \cdots x_p]) = \begin{pmatrix} F_1(x_1) \\ F_2(x_2) \\ \vdots \\ F_p(x_p) \end{pmatrix}$$

   The Jacobian matrix is given by:

$$\bar{J} = \begin{pmatrix} J_1 & & & \\ & J_2 & & \\ & & \ddots & \\ & & & J_p \end{pmatrix}$$

   **Result 3.2.3** *The number of groups required for the computation of sparse $\tilde{J}$ is less than the number of groups required by $J$ no matter what is the coloring method employed, one-sided row, one-sided column or bi-coloring.*

   To establish this result consider the case with $p = 2$ and the following products

$$\bar{J}V, \bar{J}^T W$$

   If $V, W$ are consistent with determination of $J$(one-sided row, one-sided column or bi-coloring), then $V, W$ will also be consistent with determination of both $J_1$ and $J_2$. $\Box$.

**Stacked Hessian matrices**

Since the Hessian matrix is the Jacobian of the gradient, hence all the Jacobian matrix ideas can be applied to the Hessian matrix too with regard to the gradient instead of the function $f$. Also if the function $f$ is partially separable, then so is the gradient $\nabla f$.

Assume $H = H_1 + H_2$, suppose we instead want to compute $H$ using

$$\tilde{H} = (\, H\,_1 H_2)$$

Here both $H_1$, $H_2$ are symmetric Hessian matrices. An important question to ask is: Is the computation of sparse $H$ directly cheaper than the computation of sparse $\tilde{H}$?

**Result 3.2.4** *The number of groups required for computation of sparse $\tilde{H}$ is less than the number of groups required by $H$.*

To establish this result consider the following product

$$\tilde{H}V = \begin{pmatrix} H_1 V \\ H_2 V \end{pmatrix}$$

If $V$ s consistent with determination of $H$, then $V$ will also be consistent with determination of both $H_1$ and $H_2$ since adjacency graphs of $H_1$ and $H_2$ will be subset of adjacency graph of $H_1$. $\square$.

The combinatorial problem of minimizing the numbers of groups needed for $\tilde{H}$ can be represented as an interesting graph coloring problem. Consider the adjacency graph of $H$. Classify(mark) the edges corresponding to $H_1$ as marked "1" and those corresponding to $H_2$ as marked "2". Hence there may be some edges which are marked both "1" and "2" since $H_1$ and $H_2$ might have common nonzeros. Then the graph coloring problem can be termed as: Assign colors to the nodes such that if $(v_i, v_j)$ is an edge then

- $v_i$ and $v_j$ have different colors.
- Every path of length 3 with all marked "1" or all marked "2" edges should have at least 3 colors.

This will define a consistent partition of columns and hence allow for sparse computation of $H$.

If we combine the idea of partial and variable separability, we have the Hessian matrix $\bar{H}$:

$$\bar{H} = \begin{pmatrix} H_1 & 0 \\ 0 & H_2 \end{pmatrix}$$

**Result 3.2.5** $\chi(\bar{H}) \leq \chi(\tilde{H}) \leq \chi(H)$ *where $\chi(.)$ denotes the number of groups needed to compute the sparse Hessian matrix.*

We have already proved the second part of inequality in Result 3.2.3. For the first part, if $V$ is consistent with determination of $\tilde{H}$ than $(V, V)$ is trivially consistent with determination of $\bar{H}$. $\square$.

Here are some sample results in Table 3.1 when $H_1$ and $H_2$ had about half as many elements as $H$, and $H$ was chosen with density 0.5.

Table 3.1: Sample results

| Size | number of colors for $H$ | number of colors for $\tilde{H}$ | number of colors for $\bar{H}$ |
|---|---|---|---|
| 50 | 30 | 18 | 14 |
| 100 | 71 | 51 | 31 |
| 200 | 167 | 133 | 92 |

However, these results must be taken with a pinch of salt, since the algorithms for graph coloring are heuristic in nature.

## 3.3 Relation between Structure Exploitation and Hand-coding

In this section we note the "conceptual" similarity between structured application of AD as we described it in this chapter and hand coding of derivatives. In Chapter 5, we present code templates which employ the structured automatic differentiation technology described here; if you look carefully at the code presented there is a striking similarity to hand-coded derivatives. So in a way, structured AD is an automated way of coding hand-coded derivatives.

To illustrate this similarity we consider a very simple example function:

$$f(x) = c^T e^{Ax}, \quad A \in \Re^{n \times n}, c \in \Re^n,$$

$A$ and $c$ are constants.

The "hand-coded" derivatives of this function are given by:

$$\nabla f(x) = A^T u(x), \quad \nabla^2 f(x) = A^T diag(u(x)) A, \quad u(x) = c. * e^(Ax) \tag{3.19}$$

where $diag(u(x))$ denotes a diagonal matrix with the diagonal entries from the vector $u(x)$.

Computing these "hand-coded" derivatives is cheap; both $\omega(\nabla f(x))$ and $\omega(\nabla^2 f(x))$ are proportional to one function evaluation of $f$, since the computation of $u(x)$ is equivalent to the computation of $f$, assuming that the time to multiply with $A, A^T$ can be neglected compared to a function evaluation.

Now consider the structured application of AD using extended functions; the extended function and the extended Jacobian and Hessian matrices are given by:

$$F_E(x, y) = \begin{pmatrix} y - Ax \\ c^T e^y \end{pmatrix}.$$

$$J_E = \begin{pmatrix} A & -I \\ 0 & \nabla_y(c^T e^y)^T \end{pmatrix}.$$

$$H_E = \begin{pmatrix} A & -I & 0 \\ 0 & diag(c.*e^y) & -I \\ 0 & 0 & A^T \end{pmatrix}$$

Both $\omega(J_E)$ and $\omega(H_E)$ are proportional to one evaluation of $f$, since the sparse matrices which need to be computed via AD are diagonal. The quantities we need to compute via AD are $J = diag(c.*exp(y)) \equiv diag(u(x))$ and $g^T = \nabla_y(c^T exp(y))^T \equiv u(x)^T$ (via reverse mode) which can be done in one function evaluation. The gradient and the Hessian matrix are computed using the Schur complement computation;

$$\nabla f(x) = A^T g, \quad \nabla^2 f(x) = A^T J A, \quad u(x) = c.*exp(Ax)$$

which are equivalent to hand-coded computation in equation (3.19), since $J \equiv diag(u), g \equiv u$.

## 3.4 Hierarchical Structure

$$\boxed{\begin{array}{l} \textit{Solve for } y_1: \ F_1(x, y_1) = 0 \\ \textit{Solve for } y_2: \ F_2(x, y_1, y_2) = 0 \\ \quad \vdots \\ \textit{Solve for } y_p: \ F_p(x, y_1, y_2, \ldots, y_p) = 0 \\ \textit{``Solve'' for output } z: \ z - F_{p+1}(x, y_1, y_2, \ldots, y_p) \ = \ 0 \end{array}}$$

Figure 3.7: A General Structured Computation

Figure 3.7 shows our view of general structured program as we illustrated in Section 1. We can generalise it to have the steps represent more complicated solution processes than just nonlinear equations. In general each step can represent a solution "process" e.g., parametric optimization or solution of BVPs in ODE. In general, the steps could be any solution process which can be represented in terms of nonlinear equations. This will allow the representation of the extended functions to be as high-level or abstract as possible – e.g. for problems dealing with the solution of continuous PDEs, we can have the high-level structured step represented as a continuous solve step.

Also the structure can be hierarchical as demonstrated in the example in Figure 3.8. The step 2 is shown to represent the some internal stencil structure which can be exploited hierarchically. Similarly, the step 3 is an abstract step representing the solution of a parametric optimization problem, we illusrate the stencil structure in detail in Chapter 6, and the solution of the parametric optimization problem in Chapter 7.

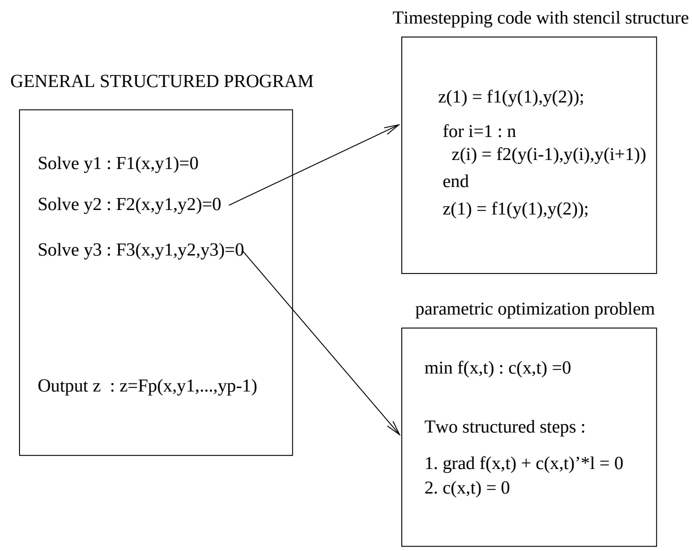



Figure 3.8: Hierarchical structure

Combining all these ideas, we can now present the completely general view of structure, where each structured step is regarded as a abstract "process" (without considering actual computation or computer code involved to implement the step) in Figure 3.9.

$$\boxed{\begin{array}{l} x \rightarrow^{\mathcal{P}_1} y_1 \;\; or \; \mathcal{P}_1(x, y_1) = 0 \\ \mathcal{P}_2(x, y_1, y_2) = 0 \\ \quad \vdots \\ \mathcal{P}_p(x, y_1, \ldots, y_p) = 0 \\ output \; z = \mathcal{P}_{p+1}(x, y_1, \ldots, y_p) \end{array}}$$

Figure 3.9: A General Structured Computation

Here the transformations $\mathcal{P}$ denote the abstract processes, just high-level representation of the computation involved. In a way, we can differentiate this abstract program to come up with the extended derivative matrix:

$$J_E \;=\; \left( \begin{array}{c|cccc} \frac{\partial \mathcal{P}_1}{\partial x} & \frac{\partial \mathcal{P}_1}{\partial y_1} & & & \\ \frac{\partial \mathcal{P}_2}{\partial x} & \frac{\partial \mathcal{P}_2}{\partial y_1} & \frac{\partial \mathcal{P}_2}{\partial y_2} & & \\ \vdots & \vdots & & \ddots & \\ \frac{\partial \mathcal{P}_p}{\partial x} & \frac{\partial \mathcal{P}_p}{\partial y_1} & \ldots & & \frac{\partial \mathcal{P}_p}{\partial y_p} \\ \hline \frac{\partial \mathcal{P}_{p+1}}{\partial x} & \frac{\partial \mathcal{P}_{p+1}}{\partial y_1} & \ldots & & \frac{\partial \mathcal{P}_{p+1}}{\partial y_p} \end{array} \right). \tag{3.20}$$

We can regard the above extended derivative computation as a process which will get us the true extended Jacobian matrix. The partials $\frac{\partial \mathcal{P}_i}{\partial y_j}$ can be considered as differentiation operators for the respective process.

Each of the processes $\mathcal{P}_i$ could also themselves be structured computations, in way illustrating the hierarchical nature of the structured computation.

It helps to think of structured computations in this way, since it captures the notion of derivative computation as a separate "derivative processes", which separates the computation of derivatives from the function computation itself – an issue which also arises in computation of sensitivities of PDE related computations, where the method to solve for the sensitivity equations(derivative process) can be totally different from the function computation.

# Chapter 4

# Matrix-Vector Operations by AD

Differentiation carried out at the level of matrix-vector operations has a lot of advantages. It reduces the spatial complexity in the reverse mode, where only high level matrix and vectors need to be saved instead of all the elementary intermediate variables. To illustrate, consider the example of dot product of two vectors: At the elementary level it is coded using a for loop:

```
function z = dotproduct(x, y)
z = 0;
for i = 1 : n
    z = z + x(i) * y(i);
end
```

The above code generates $n+1$ extra intermediate variables, which an elementary AD tool will need to deal with. However, for AD at the matrix-vector level we don't need to consider these intermediate variables as we illustrate in this chapter. This example of dot product generates a number of intermediate variables of same order of magnitude as the input arguments so the savings made aren't extraordinary, but if you consider the example of a matrix-matrix product, the number of intermediate variables generated are $n^3$, which can seriously degrade the performance of the reverse mode.

```
function C = matrixmul(A, B)
for i = 1 : m
    for j = 1 : p
        C(i, j) = 0;
        for k = 1 : n
            C(i, j) = C(i, j) + A(i, k) * B(k, j);
        end
    end
end
```

The gains made in temporal and spatial complexity (especially for the reverse mode) become more pronounced for implicit matrix-vector operations like the solution of a linear system, $y = A\backslash x$, where the number of intermediates generated is quite large. There are other insights gained by this high-level view, e.g. information about parallelization of derivative code. We present a basic treatment of parallelizing AD of matrix-vector operations in §8.

A related work in vector differentiation is included in the Ph.D. thesis by Dmitri Shiriaev [Shi93].

## 4.1 Differentiating MATLAB

### 4.1.1 Forward mode

Now we present the basic ideas involved in Automatic differentiation of a high-level language like MATLAB. We overload all the elementary functions(MATLAB builtin functions in this case, e.g. `exp`, `sum`, +, - etc.), which not only compute the "value" of the output, but also update "derivatives" of the output consistently using the chain rule to propagate Taylor series coefficients. In Table 4.1 we present a listing of how we handle some of the matrix-vector operations. $\dot{z}$ corresponds to the forward product $\frac{\partial z}{\partial I} \cdot V$, $I$ denotes the independent variables and $V$, a matrix having $p$ columns, denotes the initial tangent direction, $\dot{I} = V$. Hence $\dot{z}$ has $p$ columns.

For a vector $x \in \Re^{n\times 1}$, the forward product $\dot{x}$ is of size $n \times p$, where $p$ is the number of columns in $V$. $\dot{x}(:,i)$ denotes the derivative in the $i$th tangential direction. For a matrix $A \in \Re^{m\times n}$, the forward product $\dot{A}$ is a tensor of size $m \times n \times p$. $\dot{A}(:,:,i)$ denotes the derivative in the $i$th tangential direction.

Table 4.1: Tangent propagation rules

| Operation | Tangent Rule |
|---|---|
| $z = x^T y$ | $\dot{z} = y^T \dot{x} + x^T \dot{y}$ |
| $z = x + y$ | $\dot{z} = \dot{x} + \dot{y}$ |
| $z = x.*y$ | $\dot{z}(:,i) = \dot{x}(:,i).*y + \dot{y}(:,i).*x$ |
| $y = A*x$ | $\dot{y}(:,i) = \dot{A}(:,:,i)x + A\dot{x}(:,i)$ |
| $y = A\backslash x$ | $\dot{y}(:,i) = A\backslash(\dot{x}(:,i) - \dot{A}(:,:,i)y)$ |
| $C = A + B$ | $\dot{C} = \dot{A} + \dot{B}$ |
| $C = A*B$ | $\dot{C}(:,:,i) = \dot{A}(:,:,i)*B + A*\dot{B}(:,:,i)$ |
| $C = A.*B$ | $\dot{C}(:,:,i) = \dot{A}(:,:,i).*B + A.*\dot{B}(:,:,i)$ |
| $C = A./B$ | $\dot{C}(:,:,i) = \dot{A}(:,:,i)./B - A.*(\dot{B}(:,:,i)./(B.*B))$ |

The tangent propagation rules are easy to derive using the chain rule. The rules for the implicit matrix vector operations can be a little tricky, e.g. the "solve" operation: $y = A\backslash x$. Let us illustrate the derivation of the tangent rule by expressing the solve operation as:

$$Ay = x.$$

Differentiating w.r.t. a scalar input variable $t$

$$A\dot{y} + \dot{A}y = \dot{x}$$

or

$$\dot{y} = A \backslash (\dot{x} - \dot{A}y).$$

This can be generalised to differentiation w.r.t. a vector to give the rule in Table 4.1.

### 4.1.2 Reverse mode

Now we present the rules for propagation of adjoints. Again all the elementary functions are overloaded for this purpose. In Table 4.2 we present a listing of how we handle the computation of adjoints for some of the matrix vector operations. $z^*$ corresponds to the adjoint product $\frac{\partial O}{\partial z}^T \cdot W$, $O$ denotes the output variables and $W$ denotes the initial adjoint direction, $\dot{O} = W$.

For a vector $x \in \Re^{n \times 1}$, the adjoint $x^*$ is of size $n \times p$, where $p$ is the number of columns in $W$. $x^*(:, i)$ denotes the derivative in the $i$th adjoint direction. For a matrix $A \in \Re^{m \times n}$, the adjoint $A^*$ is a tensor of size $m \times n \times p$. $A^*(:, :, i)$ denotes the derivative in the $i$th adjoint direction.

Table 4.2: Adjoint propagation rules

| Operation | Adjoint Rule |
|---|---|
| $z = x^T y$ | $x^* = y * z^*, \quad y^* = x * z^*$ |
| $z = x + y$ | $x^* = z^*, \quad y^* = z^*$ |
| $z = x. * y$ | $x^* = diag(y) * z^*, \quad y^* = diag(x) * z^*$ |
| $y = A * x$ | $x^* = A^T * y^*, \quad A^*(:, j, :) = x(j) * y^*$ |
| $y = A \backslash x$ | $x^* = A^T \backslash y^*, \quad A^*(:, :, i) = -(A^T \backslash y^*(:, i))y^T$ |
| $C = A + B$ | $A^* = C^*, \quad B^* = C^*$ |
| $C = A * B$ | $A^*(:, :, i) = B^T * C^*(:, :, i), \quad B^*(:, :, i) = A^T * C^*(:, :, i)$ |
| $C = A. * B$ | $A^*(:, :, i) = C^*(:, :, i). * B, \quad B^*(:, :, i) = C^*(:, :, i). * A$ |
| $C = A./B$ | $A^*(:, :, i) = C^*(:, :, i)./B, \quad B^*(:, :, i) = -C^*(:, :, i). * A./(B. * B)$ |

Reverse mode at this high-level saves considerable amount of space complexity, in operations $x^T y$, $Ax$, $A \backslash x$, $A * B$ since it avoids the temporary variables formed in these computations. Particularly in the operations involving $n \times n$ matrices, the reduction can be an order of magnitude.

The adjoint propagation rules shown in Table 4.2 are easy to derive except for those involving the implicit matrix vector operations can be a a little difficult to derive, e.g. the "solve" operation: $y = A \backslash x$. Adjoint $x^* = \frac{\partial O}{\partial x}^T \cdot W = \frac{\partial y}{\partial x}^T y^*$ can be derived easily by noticing that $\frac{\partial y}{\partial x} = A^{-1}$, hence $x^* = A^T \backslash y^*$ which involves a solution in $A^T$. Computation of the adjoint of $A$ is a little tricky, and can be derived using the following recipe. Differentiating $Ay = x$ w.r.t. the scalar $A(i,j)$ yields:

$$A\partial y + \partial A(i,j)y_j * e_i = 0$$

or

$$\frac{\partial y}{\partial A(i,j)} = -A\backslash(e_i * y_j).$$

The adjoint of $A(i,j)$ can be now written as

$$A^*(i,j) = -\frac{\partial y}{\partial A(i,j)}^T y^* = -(A^{-1}(e_i * y_j))^T y^* = -(e_i * y_j)^T (A^T\backslash y^*)$$

By covering all indices $(i,j)$, we come up with the formula presented in the Table 4.2, i.e. $A^* = -(A^T\backslash y^*)y^T$.

### 4.1.3 Sparsity pattern computation

One of the major functions of ADMAT tool is to compute the sparsity patterns of Jacobian and Hessian matrices automatically. The sparsity pattern of Jacobian matrix can be propagated exactly the tangent products. In Table 4.3 we provide a listing of the rules for sparsity pattern propagation similar to the propagation of forward products.

Assume that the size of the independent vector $I$ is $n \times 1$. For an intermediate vector $v \in \Re^{n_v \times 1}$, the Jacobian sparsity pattern $J_v$ is of size $n_v \times n$. For an intermediate matrix $A \in \Re^{m_A \times n_A}$, the Jacobian sparsity pattern $J_A$ is a tensor of size $m_A \times n_A \times n$. The sparsity pattern of a variable is propagated as a 0-1 vector, matrix or a tensor depending on the size of the variable involved. Since the sparsity pattern is 0-1 vector, matrix or tensor, the sparsity pattern rules have to be carried out in 0-1 arithmetic, i.e. $1+1=1$, $0*1=0$, $0+1=1$, $0+0=0$ and so on. After each operation we just take the sparsity pattern of the resuting matrix of tensor as the resulting 0-1 matrix or tensor.

#### Computing Hessian sparsity pattern

The sparsity pattern of Hessian matrix is slightly more complex. Here we need to propagate the sparsity patterns of Jacobians (1st order derivatives) together with the Hessian sparsity patterns (2nd order derivatives). Hence we need chain rule propagation from 2nd order Taylor series.

In table 4.4 a listing of the rules for the propagation of Hessian sparsity patterns. Here $H_z$ denotes the sparsity pattern of Hessian of $z$, and $J_z$ denotes the sparsity pattern of the Jacobian (gradient) of $z$. Propagation of $J_z$ is governed as specified in the previous section.

Table 4.3: Jacobian sparsity pattern propagation rules

| Operation | Sparsity pattern Rule |
|---|---|
| $z = x^T y$ | $J_z = \sum J_{x(i)} + J_{y(i)}$ |
| $z = x + y$ | $J_z = J_x + J_y$ |
| $z = x. * y$ | $J_z = J_x + J_y$ |
| $y = Ax$ | $J_y(i) = \sum J_{A(i,:))} + A(i,:) * J_x$ |
| $C = A + B$ | $J_C = J_A + J_B$ |
| $C = A * B$ | $J_C = J_A * B + A * J_B$ |
| $C = A. * B$ | $J_C = J_A + J_B$ |
| $C = A./B$ | $J_C = J_A + J_B$ |

Table 4.4: Hessian sparsity pattern propagation rules

| Operation | Sparsity pattern Rule |
|---|---|
| $z = x + y$ | $H_z = H_x + H_y$ |
| $z = x. * y$ | $H_z = H_x + H_y + J_x J_y^T + J_y J_x^T$ |
| $C = A + B$ | $H_C = H_A + H_B$ |
| $C = A. * B$ | $H_C = H_A + H_B + J_A J_B^T + J_B J_A^T$ |
| $C = A./B$ | $H_C = H_A + H_B + J_A J_B^T + J_B J_A^T$ |

## 4.2 ADMAT – AD Tool for MATLAB

ADMAT enables you to differentiate target functions in MATLAB and implements both forward and reverse modes of automatic differentiation. This tool belongs to the "operator overloading" class of AD tools and uses MATLAB5's OOP (Object Oriented Programming) feature for implementation.

The tool uses the rules presented in the previous section to implement the propagation of derivatives. The core of the AD tool is the MATLAB class, deriv, which implements the forward mode of automatic differentiation. This class provides method to compute the forward mode product $JV$, given any nonlinear function $F$ written in Matlab.

### Basic forward mode

Here is the basic structure of class deriv. It consists of two fields, namely value and deriv which stand for the value of the variable and the derivative. All the elementary functions (MATLAB built-in functions in this case, e.g. exp, sum, +, - etc) are overloaded for the deriv class, which not only compute the "value" of the output, but also update the "deriv" of output consistently using chain rule to propagate the Taylor series coefficients. This is in accordance with the rules shown in Table 4.1

```
class @deriv {
double value;
double deriv;
}
```

## Basic reverse mode

The class derivtape built on top of deriv implements the basic reverse mode and provides methods to compute the reverse mode product $W^T J$. This class also consists of two fields, namely value which stands for the value of the variable and varcount which is a unique number for every intermediate variable in the computation. This number is used to maintain the trace (or "tape") for the function. To implement the reverse mode, the AD tool implements a tape, which records **all** the intermediate values and operations performed in the function evaluation. The computation of adjoints is done by a reverse pass on the tape; the tape is described in the next subsection.

```
class @derivtape {
double value;
double varcount;
}
```

## Combined mode for Hessian matrix product

The class derivtapeH is used to compute the Hessian matrix product, since computing $HV$ is the same as the reverse mode on the forward product $\nabla f^T V$. So this class employs the functionality of the basic forward mode and reverse mode classes, by first computing the function as well as $\nabla f^T V$ using the forward mode (deriv class) and then uses the reverse mode to compute the gradient of $\nabla f^T V$.

To implement this combined behavior, this class has two fields one for the value of the intermediate $z$, and other for the the forward product $\nabla z^T V$. Both these fields are reverse mode enabled, i.e. they belong to the derivtape class.

```
class @derivtapeH {
@derivtape value;
@derivtape deriv;
}
```

## Sparsity pattern mode

### Sparsity pattern of the Jacobian matrix

For computing the sparsity pattern of the Jacobian, the AD tool uses a different class called derivspj. This time the sparsity pattern of the gradient of each intermediate value is propagated using the methods in derivspj.

```
class @derivspj {
        double value;
        double SPJ;
}
```

### Sparsity pattern of the Hessian matrix

Class derivsph is used for computing the sparsity pattern of the Hessian. It builds on derivspj. The sparsity pattern of the gradient as well as the Hessian of each intermediate value is propagated using the methods in derivsph.

```
class @derivsph {
        @derivspj SPJ;
        double SPH;
}
```

## The ADMAT "tape" or the computational graph

For the reverse propagation of derivatives, the whole execution trace of the original evaluation program must be recorded, unless it is recalculated as illustrated in [Gri92a]. In ADMAT, this potentially huge data set is written into a MATLAB structure which is referred to as the **tape**.

In ADMAT, the tape is designed as follows. The tape contains the complete execution trace of the computation. The tape is a computation graph, with a node corresponding to every intermediate variable. A node has the following fields:

1. **Op:** Stands for the arithmetic operation which generated this intermediate variable.

2. **Val:** Value of this intermediate variable.

3. **Arglist:** Pointers to nodes (other intermediate or input variables involved in computation of this variable).

4. **deriv:** This field contains the associated derivative information of this intermediate variable. In forward mode computation (when using the tape), this will be the intermediate Jacobian-matrix product $JV$, and in the reverse mode this field will contain the adjoint.

Figure 4.1 shows an example tape for the sample function:

*function* $y$ = *getfun*$(x)$
    $z = x * x;$
    $z = x + z;$
    $y = z * z;$

ADMAT includes methods to do reverse sweeps on the tape to compute the adjoint product.

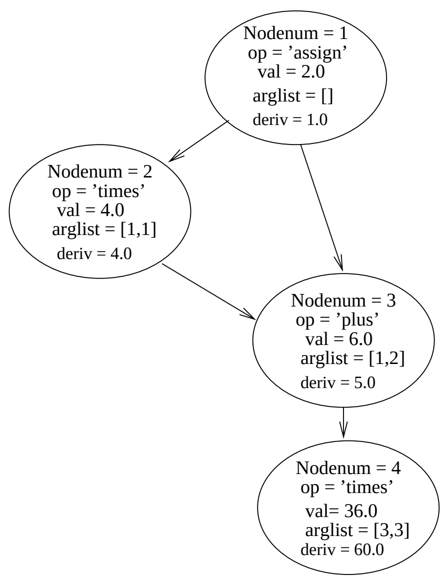



Figure 4.1: tape corresponding to a simple program

# Chapter 5

# Semi-Automatic Differentiation

Effective use of AD software for realistic large-scale problems is often not "automatic." Indeed, performance gains of several orders of magnitude can sometimes be achieved by using AD in a selective manner (as opposed to straightforward use of AD software). In particular, large-scale problems typically exhibit structure: for AD to be used efficiently (or even feasibly) it is crucial that the dominant problem structure be understood and exploited. This is certainly true for optimal design (and inverse) problems. Direct application of AD can be unbearably expensive on such problems due to their density and complexity. On the other hand, if the use of AD is integrated with the problem structure then there is significant potential for effective calculation of derivatives using AD software. This kind of application of AD is what we call "semi-automatic" differentiation.

Semi-automatic differentiation involves a very special role from the user. Unless the problem structure is automatically detected, the user must indicate the problem structure. ADMIT-2 software, which is presented briefly in Appendix C, is designed to use the problem related structure information provided by the user and uses the EASE techniques to exploit the structure present in the problem. Often, the problem structure present in specific applications is best exploited by structure exploitation techniques applied to specific code segments in the computation. The stencil computation presented in Chapter 6, is one example of applying AD to only certain code segments to achieve efficiency.

However, semi-automatic differentiation comes with a cost. The user needs to be educated about coding the "derivative code" which applies AD to parts of the original code. In this chapter we present guidelines and some examples which will help user to write derivative code using a standard AD tool.

## 5.1 Basics of Writing "Derivative Code"

In this section we present some notation and guidelines for writing templates of derivative code using AD technology, which then can be used write specific structure exploitation codes for different applications.

Typically, structure exploitation means applying black-box AD to only a part of the computation intelligently. To understand the writing of these templates, we need to first understand how the tangent and adjoint derivative propagations work. The code templates presented here form the backbone of the software technology of ADMIT-2 (refer to appendix C.3).

We have shown how derivative propagation works at the basic computer arithmetic level in an automatic differentiation tool in Chapter 1. In this section, we look at the propagation of derivatives in more detail and at a high level representation of the function.

### 5.1.1 Propagation of tangents

Let's consider a general nonlinear function $g = F(x)$, where $F \in \Re^n \to \Re^m$. Typically the computation of $F$ via a computer procedure might involve a set of intermediates variables. Here $g$ denote the vector of output variables and $x$ denote the vector of input variables. Let us associate with each variable $z$ in the program (intermediate, input or output), a tangent $z'$ which is defined by

$$z' = \frac{dz}{dx} * V \tag{5.1}$$

$V$ denotes the input (or user-supplied) tangent of the input variable $x$ since by definition (equation (5.1)):

$$x' = \frac{dx}{dx} V = V$$

The tangents can be propagated along with the computation. The desired quantity to compute is $g'$, the tangent of the dependent variable $g$. Using equation (5.1) $g'$ is given by:

$$g' = \frac{dg}{dx} * V = J * V$$

Here $J$ denotes the Jacobian matrix of the mapping $F$.

Here is a recipe to propagate the tangents through the computation. If an intermediate (or output) variable $z$ depends on a previous intermediate (or input) variable $u$ via some intermediate function $z = f(u)$, the tangent of $z$ is given by:

$$\begin{aligned}
z' &= \frac{\partial z}{\partial x} V \\
&= (\frac{\partial f}{\partial u} \frac{\partial u}{\partial z}) * V \\
&= \frac{\partial f}{\partial u} \frac{\partial u}{\partial x} V \\
&= \frac{\partial f}{\partial u} (\frac{\partial u}{\partial x} V).
\end{aligned}$$

By using the definition of tangent of variable $u$, i.e., $u' = \frac{\partial u}{\partial x} V$ we can now write:

$$z' \;=\; \frac{\partial f}{\partial u} * u'. \tag{5.2}$$

However, variables other than $u$ may be used to compute $z$, i.e. there can be more than arguments to $f$, so the above calculation is just one part of the tangent of $z$. To compute the full tangent of $z$, all the arguments should be taken into account. Assume that $z = f(u_1, u_2, \ldots, u_p)$. Then the chain rule can be written as:

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{p} \frac{\partial f}{\partial u_i} \frac{\partial u_i}{\partial x},$$

and hence the tangent of $z$ will be given by

$$z' = \sum_{i=1}^{p} \frac{\partial f}{\partial u_i} u'_i$$

by generalising equation (5.2).

### 5.1.2 Propagation of adjoints

Let us associate with each variable $z$ in the program (intermediate, input or input), an adjoint $z^*$ which is defined by

$$z^* = \frac{dg}{dz}^T W. \tag{5.3}$$

$W$ denotes the input (or user-supplied) adjoint of the output variable $g$ since by definition (equation (5.3)):

$$g^* = \frac{dg}{dg}^T W = W.$$

The desired quantity to compute is $x^*$, the adjoint of the independent variable $x$. To compute $x^*$, we need to go through the computation in the reverse order to propagate the adjoints. Using equation (5.3) $x^*$ is given by:

$$x^* = \frac{dg}{dx}^T W = J^T W.$$

Here $J$ denotes the Jacobian matrix of the mapping $F$.

Here is a recipe to propagate the adjoints backwards through the computation. If an intermediate(or output) variable $z$ depends on a previous intermediate (or input) variable $u$ via some intermediate function $z = f(\ldots, u, \ldots)$, and this is the only occurence of variable $u$, then the adjoint of $u$ is given by:

$$\begin{aligned} u^* &= \frac{\partial g}{\partial u}^T W \\ &= (\frac{\partial g}{\partial z}\frac{\partial f}{\partial u})^T W \\ &= \frac{\partial f}{\partial u}^T \frac{\partial g}{\partial z}^T W \\ &= \frac{\partial f}{\partial u}^T (\frac{\partial g}{\partial z}^T W) \end{aligned}$$

by using the chain rule:

$$\frac{\partial g}{\partial u} = \frac{\partial g}{\partial z}\frac{\partial f}{\partial u}.$$

By using the definition of adjoint of variable $x$, i.e., $x^* = \frac{\partial g}{\partial x}^T W$ we can now write:

$$u^* = \frac{\partial f}{\partial u}^T z^* \tag{5.4}$$

However, $u$ may be used to compute more than one intermediate (or output) variables, so the above calculation is just one part of the adjoint of $u$. To compute the full adjoint of $u$, all these contributions should be summed over all the variables $u$ is directly used in computing. Assume that $u$ is directly used in computed $p$ intermediate(or output) variables $z_1, \ldots, z_p$ via functions $f_i$, i.e. the computation of $z_i$ uses $u$ directly: $z_i = f_i(\ldots, u, \ldots)$, then chain rule can be written as:

$$\frac{\partial g}{\partial u} = \sum_{i=1}^{p} \frac{\partial g}{\partial z_i}\frac{\partial f_i}{\partial u},$$

and hence the adjoint of $u$ will be given by

$$u^* = \sum_{i=1}^{p} \frac{\partial f_i}{\partial u}^T z_i^*$$

by generalising equation (5.4).

Hence we can compute the adjoints of any computation by using the above rules of propagating the adjoints in a reverse order through the computation.

## 5.2 Coding the Derivatives of Structured Computation

In this section we demonstrate the writing of tangent and adjoint product codes for structured computations. We demonstrate writing these code templates for the most general case of a structured computation.

We consider a restricted version of general structured program for clarity, i.e. as shown in Figure 5.1:

Figure 5.2 shows the MATLAB-like code which computes the above function:

“Solve” for $y_1$ : $y_1 = F^1(x)$
“Solve” for $y_2$ : $y_2 = F^2(x, y_1)$
$\vdots$
“Solve” for $y_k$ : $y_k = F^k(x, y_1, y_2, \ldots, y_{k-1})$
“Solve” for output $z$ : $z = F^{k+1}(x, y_1, y_2, \ldots, y_k)$

Figure 5.1: A (restricted) general structured computation

```
for i = 1 : k
    y_i = F_i(x, y_1, ..., y_{i-1});
end
z = F_{k+1}(x, y_1, ..., y_k);
```

Figure 5.2: Algorithm to compute the function

## Computing tangent products

Figure 5.3 shows the template to compute the tangent product for the general computation (Figure 5.1). Here $V$ denotes the initial tangent direction. The variable $V_i$ denotes the tangent for input arguments of the function $F_i$.

```
V_1 = V;
for i = 1 : k
    y_i = F_i(x, y_1, ..., y_{i-1});
     compute JV_i = J_i * V_i; using AD.
    V_{i+1} = [V_i; JV_i];
end
compute z = F_{k+1}(x, y_1, ..., y_k);
compute JV = J_{k+1} * V_{k+1} using AD
```

Figure 5.3: Algorithm to compute $JV$

## Computing adjoint products

The adjoint product computation marches backwards through the computation and requires saving all the intermediate values, as well as saving all intermediate products of the form $J_i^T W_i$. The algorithm is shown in Figure 5.4.

We have provided the templates for computing the tangent and the adjoint product for the general structured program. We also might require computing the Newton step, or the full Jacobian; hence an important computation is the computation of extended derivative matrix itself. The extended Jacobian matrix allows for computing the Newton step and the full Jacobian matrix. Here we present the template to compute the extended Jacobian matrix of the above general computation.

```
for i = 1 : k
    y_i = F_i(x, y_1, ..., y_{i-1});
    save y_i;
end
W_{k+1} = W;
for i = k+1 : -1 : 1
    compute JW_i = J_i^T W_i using AD.
    W_{i-1} = sum_{j=i}^{k+1} JW_j(i, :);
end
JW = W_0;
```

**for** $i = 1 : k$
  $y_i = F_i(x, y_1, \ldots, y_{i-1})$;
  *save* $y_i$;
**end**
$W_{k+1} = W$;
**for** $i = k+1 : -1 : 1$
  *compute* $JW_i = J_i^T W_i$ *using AD.*
  $W_{i-1} = \sum_{j=i}^{k+1} JW_j(i,:)$;
**end**
$JW = W_0$;

Figure 5.4: Algorithm to compute $W^T J$

$ExtJ = []$;
**for** $i = 1 : k$
  $y_i = F_i(x, y_1, \ldots, y_{i-1})$;
  *compute* $J_i$ *using sparse AD.*

$$ExtJ = \begin{pmatrix} ExtJ & 0 \\ J_i & -I \end{pmatrix}$$

**end**
*compute* $z = F_{k+1}(x, y_1, \ldots, y_k)$;
*compute* $J_{k+1}$ *using sparse AD*
$ExtJ = [ExtJ; J_{k+1}]$;

Figure 5.5: Algorithm to compute the extended Jacobian matrix

# Chapter 6

# Structured Solution of Inverse Problems

This chapter deals with the use of structure exploitation technology in the general area of inverse problems. The class of inverse problems captures many aspects of "structure" that we have outlined and in our opinion it is a very important and wide application area which needs a special treatment. The need of a practical solution strategy for large-scale inverse problems also motivated the discovery of some special instances of structure, e.g. the exploitation of the stencil structure which we present in this chapter.

This chapter includes examples of four inverse problems coming from four different areas of computational science. The examples we include here include a design problem in heat conductivity, a reflection seismology inverse problem, a medical imaging inverse problem and finally an option pricing inverse problem in finance.

We thank Fadil Santosa of University of Minnesota for introducing us to the inverse problems in heat conductivity and wave propgation settings. The exploitation of the stencil structure is joint work with him.

## 6.1 Inverse Problems

Inverse problems are typically viewed as large non-linear data fitting problems. The desired solutions are estimates of certain parameters in the model governing a process, e.g. a physical, chemical or financial process.

A large number of engineering design problems are posed as inverse problems. These engineering design problems are typically based on a *forward* computation that can be viewed as follows. Given the value of parameters $x = \bar{x} \in \Re^n$, solve for $y \in \Re^m$,

$$F(\bar{x}, y) = 0 \tag{6.1}$$

where the function $F$ is typically nonlinear and is assumed to be differentiable. Often $F$ represents a numerical finite difference method for the approximate solution of a differential equation. The parameter set $x$ typically consists of boundary values or other design variables and $y$ is the solution. Typically, $m \geq n$, and $F$ is a square system w.r.t. $y$.

Now the *inverse* problem corresponding to this forward computation can be phrased as follows. Given a target $\bar{y}$, determine values for the parameters $x$ such that the forward computation yields a value *close* to the target $\bar{y}$.

There are two popular ways of solving the above problem, in the case $m = n$, we look for possibly an exact match by solving the nonlinear equation problem,

$$y(x) - \bar{y} = 0 \tag{6.2}$$

and when $m > n$ or when the target data $\bar{y}$ may be inaccurate, we look for a "close" match by solving the nonlinear least squares problem,

$$\min \quad \|y(x) - \bar{y}\|_2. \tag{6.3}$$

where $y(x)$ is implicitly defined by the forward process (6.1).

Now we start with a presentation of a canonical inverse problem in heat conductivity and present a detailed solution using structure exploitation and AD.

## 6.2 Heat Conductivity Inverse Problem

To illustrate the structured application of AD in a concrete way, we consider a very simple inverse problem involving heat transfer. The problem is to find conductivity properties of a 1-dimensional bar whose predicted temperature evolution matches desired (or measured) behavior.

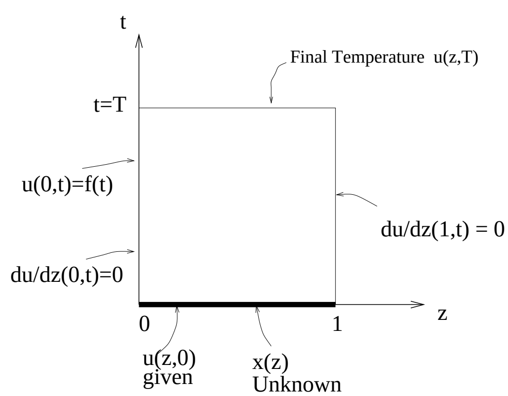



Figure 6.1: 1-D inverse problem for the heat equation

The setup of the 1-D heat equation is shown in Figure 6.1. There is a thin bar with ends at $z = 0$ and $z = 1$. The governing partial differential equation is shown:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial z}(x(z)\frac{\partial u}{\partial z}). \tag{6.4}$$

The function $u(z,t)$ represents the temperature of the bar at position $z$ and time $t$. The function $x(z)$ represents the *unknown* conductivity of the bar. We aim to determine $x(z)$, to closely match the measured behavior. Suppose that the initial temperature distribution $u(z,0)$ is known and both ends of the bar are insulated while the left end temperature is prescribed; the temperature over the bar is allowed to evolve based on equation (6.4). The target (or measured) temperature distribution over the bar $\phi_{tar}(z)$ is specified at time $t = T$. Now we wish to solve for the conductivity function $x(z)$ which results in a close match between the target temperature $\phi_{tar}(z)$ and the model temperature distribution $\phi(z) = u(z,T)$. A related inverse problem in heat transfer is described in Mukherjee et al [ZMR88]. The problem of solving for $x(z)$ is phrased as a least-squares problem:

$$\min_{x(z)} \|\phi_{tar}(\cdot) - u(\cdot, T)\|_2.$$

Computationally, this problem is solved using a discretization of spatial and time domains, and employing a suitable finite difference method. For example we can use the following discretization:

$$z_j = (j-1)\Delta z, j = 1 : N, \Delta z = \tfrac{1}{N-1}$$
$$t_k = k\Delta t, k = 0 : M, \Delta t = \tfrac{T}{M}$$
$$u_j^{(k)} = u(z_j, t_k)$$

One suitable method for solving the 1-D heat equation is the following :

$$\frac{u_j^{(k+1)} - u_j^{(k)}}{\Delta t} = \frac{1}{2}(D_+(x(z_j)D_-(u)) + D_-(x(z_j)D_+(u))).$$

Here $D_+, D_-$ are the spatial difference operators. Expanding:

$$u_j^{(k+1)} = c_j u_j^{(k)} + c_{j+1} u_{j+1}^{(k)} + c_{j-1} u_{j-1}^{(k)}. \tag{6.5}$$

After taking the boundary conditions into account, the computation together can be written in a vector form as

$$u^{k+1} = K(x)u^k + h^k,$$

where $u^k$ denotes the discretized temperature at time $t = k\Delta t$, i.e.,
$u^k = [u_1^{(k)}, \ldots, u_N^{(k)}]$.

The matrix $K(x)$ is a tridiagonal matrix; for details see [CSV97]. The inverse problem is typically solved by solving the nonlinear equation:

$$F(x) = u^M(x) - \phi_{tar} = 0.$$

### 6.2.1 Exploiting structure in computation

Define the function $u^+ = G(x,u) = K(x)u + h$. If we apply EASE, the "structured" computation of $F(x)$ in 1-D heat equation can be written as shown in Figure 6.2.

*Solve for* $u^1$ : $u^1 = G(x, u^0)$;
*Solve for* $u^2$ : $u^2 = G(x, u^1)$;
$\vdots$
*Solve for* $u^M$ : $u^M = G(x, u^{M-1})$;
*Compute* $F := u^M - \phi_{tar}$.

Figure 6.2: Heat equation extended function

Differentiating, we obtain the extended Jacobian given by:

$$J_E = \left( \begin{array}{c|cccccc} G_x(x,u^0) & -I & & & & & \\ G_x(x,u^1) & K(x) & -I & & & & \\ \vdots & & \ddots & \ddots & & & \\ G_x(x,u^{M-2}) & & & K(x) & -I & & \\ G_x(x,u^{M-1}) & & & & K(x) & -I & \\ \hline 0 & & & & & I & \end{array} \right).$$

Using the above formulation, we get all the benefits provided by EASE. In particular, the Newton step $\Delta x = -J^{-1}(u^M(x) - \phi_{tar})$ can be computed very cheaply.

### 6.2.2 Numerical results

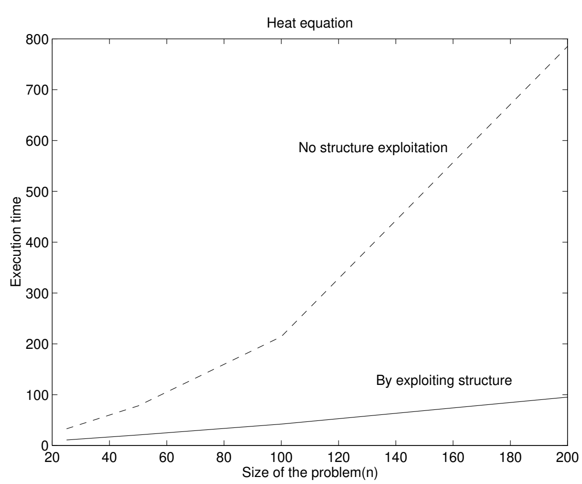



Figure 6.3: Numerical results for heat equation problem

The graph in Figure 6.3 shows the effectiveness of EASE compared to the unstructured way of computing the Jacobian and Newton step. The size of the problem is $N$, the number of points in the discretization of the spatial interval $[0, 1]$. It is easy to show that the unstructured method is quadratic in the dimension of the

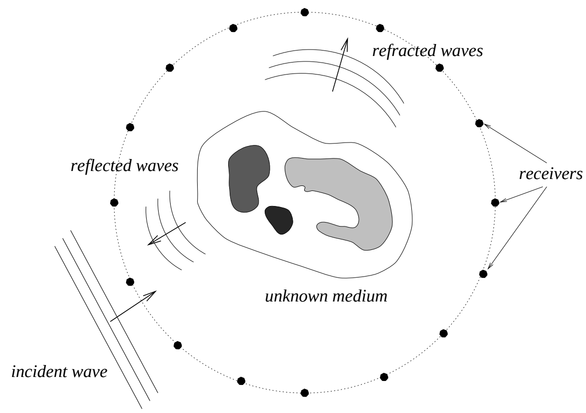



Figure 6.4: Wave propagational inverse problem setup. In this figure, the problem is to identify the unknown medium. An incident wave is generated, and as it travels into the medium being probed, reflected and refracted signals are generated. These are captured in the receivers. The inverse problem is to find the properties of the unknown medium from the collected data.

problem, but the EASE method is linear. Hence we can show an order of magnitude improvement over the dense, unstructured method. In particular, we can show that the unstructured method is $O(MN^2)$ in complexity, while the EASE scheme is $O(MN)$ in complexity. For the purpose of this experiment, the sparse solve on the extended Jacobian matrix was done using the MATLAB's backslash (\) operator. ADMIT-2 software, which is described in Appendix C, was employed.

## 6.3 Wave Propagational Inverse Problems

Wave propagational inverse problems arise in several applications including medical imaging and geophysical exploration. In the type of wave propagational inverse problems under consideration, the goal is to determine parameters, such as sound-speed distribution and density distribution, from measured data, which are collected at a set of receivers. Figure 6.4 explains the situation. An incident disturbances is generated, as it travels in the unknown medium and produces reflections and refractions. The inverse problem is to determine properties of the unknown medium from the set of measured response.

Problems of this type arise in several applications including geophysical exploration and medical imaging. A common feature in these applications is that the problem is very large. Typically, the number of unknowns and equations is in the range of $10^3$ to $10^6$. Often, the most convenient way to solve this type of inverse problem is to pose it as an optimization problem, either using a nonlinear least-squares or other approach specialized to take advantage of the properties afforded by the particular application [SS88].

We now consider a simplified setup of the reflection seismology problem as shown in Figure 6.5. The desired solutions in this case are the sound speeds at various positions under the earth's surface and the data re the reflection seismograms collected at the surface.

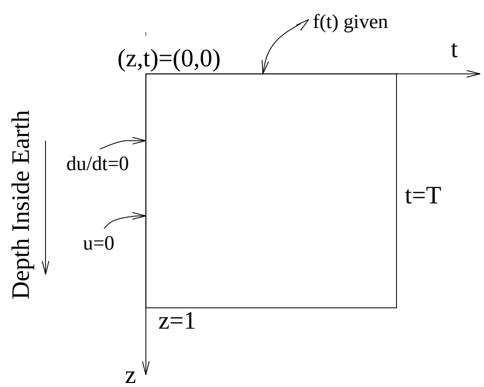



Figure 6.5: Reflection seismology problem

The problem can be described by the governing PDE (a wave equation) shown below:

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial z}(c(z)\frac{\partial u}{\partial z}). \tag{6.6}$$

The boundary conditions are as follows :

$$u(z,0) = 0, \tag{6.7}$$

$$\frac{\partial u}{\partial t}(z,0) = 0, \tag{6.8}$$

$$c(0) \cdot \frac{\partial u}{\partial z}(0,t) = f(t). \tag{6.9}$$

Here $z$ denotes the depth co-ordinate, and $c(z)$ relates directly to sound speeds at depth $z$ inside the earth. The function $u(z,t)$ represents the medium particle displacement at depth coordinate $z$ and time $t$. The function $f(t)$ denotes the excitation force, in the form of the traction applied at the surface ($z = 0$).

Discretizing the time and the spatial domains, we obtain a suitable finite element method in the form:

$$u^{k+1} = -u^{k-1} + A(c)u^k + h^k,$$

where the matrix $A(c)$ is a tridiagonal matrix.

The first iteration can be started by using the boundary condition $u^0 = 0$, also $u^{-1}$ is assumed to be identically zero.

The measurements are made at the earth's surface ($z = 0$), i.e., the measured data is the vector $[u^1(0), u^2(0), \ldots u^M(0)]$, which can be represented by $g = e_1 e_1^T u^1 + e_2 e_1^T u^2 + \ldots + e_M e_1^T u^M$. We assume that $M \geq N$ to avoid an underdetermined system for the solution of $c$.

If $M = N$, the inverse problem is solved by solving the nonlinear equation

$$F(c) = g(c) - g_{desired} = 0,$$

but if $M > N$, we look for a solution which minimizes the error by solving the least-squares problem

$$\min_{c} \quad f(c) = \|g(\cdot) - g_{desired}\|_2 \tag{6.10}$$

We shall discuss only the nonlinear equation setting in this section. For an illustration of the application of EASE for a nonlinear least squares setting, please refer to the appendix.

### 6.3.1 Exploiting structure in computation

Define the function $u^+ = F(c, u, u^-) = -u^- + A(c)u + h$. Applying EASE, we see that the extended function can be written as shown in Figure 6.6.

| |
|---|
| *Solve for* $u^1$ : $u^1 = F(c, u^0, u^{-1})$; |
| *Solve for* $u^2$ : $u^2 = F(c, u^1, u^0)$; |
| $\vdots$ |
| *Solve for* $u^M$ : $u^M = F(c, u^{M-1}, u^{M-2})$; |
| *Final* $g = e_1 e_1^T u^1 + e_2 e_1^T u^2 + \ldots + e_M e_1^T u^M - g_{desired}$. |

Figure 6.6: Wave equation extended function

The extended Jacobian is given by :

$$J_E = \left(\begin{array}{c|cccccc} F_c(c, u^0, u^{-1}) & -I & & & & & \\ F_c(c, u^1, u^0) & A(x) & -I & & & & \\ \vdots & \ddots & \ddots & \ddots & & & \\ F_c(c, u^{M-2}, u^{M-3}) & & -I & A(x) & -I & & \\ F_c(c, u^{M-1}, u^{M-2}) & & & -I & A(x) & -I & \\ \hline 0 & e_1 e_1^T & e_2 e_1^T & \cdots & & & e_M e_1^T \end{array}\right). \tag{6.11}$$

## 6.4 Exploiting Stencil Structure

Consider the seismic inverse problem from the previous section. The complete forward problem can be coded using the following timestepping scheme :

$$u^{k+1} = F(c, u^k, u^{k-1}).$$

We use the code templates provided in Chapter 5 and use semi-automatic differentiation at the timestep level. For the solution of a least squares problem in equation (6.10), the gradient of the sum of squares function needs to be calculated, which is an adjoint computation. We can differentiate the timestep computations in the reverse order to propagate the adjoints all the way back to timestep $k = 1$. We look at a general step $k$ :

$$u^{k+1} = F(c, u^k, u^{k-1})$$

The adjoints are updated by:

$$\begin{aligned}(u^k)^* &= u^k * + (\frac{\partial F}{\partial u^k})^T (u^{k+1})^* \\ (u^{k-1})^* &= u^{k-1} * + (\frac{\partial F}{\partial u^{k-1}})^T (u^{k+1})^* \\ c^* &= c^* + (\frac{\partial F}{\partial c})^T (u^{k+1})^*\end{aligned}$$

In the next subsection we present pseudocodes to do adjoint computations of this problem using the method presented here.

### 6.4.1 Code templates for the forward problem and the adjoint

We first present the forward computation which is shown in Figure 6.7. The vector $u$ represents the current state and $c$ represents the set of unknowns (e.g. sound speeds for the seismology problem). $K$ denotes the number of time steps and $n$ denotes the size of a state $u$.

```
K = 100; n = 50; c = given;
u0 = zeros(n, 1); u1 = zeros(n, 1);
g = zeros(K, 1);
for k = 1 : K
    u = timestep(c, u1, u0);
    g(k) = u(1);
    u0 = u1; u1 = u;
end
```

Figure 6.7: The 1-D complete forward computation

The adjoint code is shown in Figure 6.8.

For large problems like this, computing adjoint product of the timestep routine using basic automatic differentiation can be very expensive, since the size of the state $u$ is large and consequently the `timestep` routine will be very compute intensive. For example, ADOL-C will start writing tapes on the disk once the timestep computation becomes very big.

This concern brings us to the idea of AD of stencil codes. As we mentioned here, for large-scale problems, Automatic differentiation of the whole timestep routine can be expensive, especially in the reverse (or adjoint) mode where the AD tool needs

```
% Save all the states using a forward pass
adju = zeros(n, K + 2); adjc = zeros(n, 1);
% Initialise adjoints
adjg = ones(K, 1);
for k = K : -1 : 1
% Recover The states u1, u0
    adju(1, k + 2) = adju(1, k + 2) + adjg(k);
     % Compute and Update the adjoints
    adju(:, k) = adju(:, k) + (∂timestep(c,u1,u0)/∂u0)^T adju(:, k + 2);
    adju(:, k + 1) = adju(:, k + 1) + (∂timestep(c,u1,u0)/∂u1)^T adju(:, k + 2);
    adjc = adjc + (∂timestep(c,u1,u0)/∂c)^T adju(:, k + 2);
end
```

Figure 6.8: The 1-D complete adjoint computation (full timestep)

to store all intermediate values in order to prepare for an adjoint sweep[1]. However, it pays to expose the stencil structure of the computation, and do AD at a lower level, i.e. that of stencils. In the next section we describe how to differentiate stencil codes.

### 6.4.2 Exploiting the stencil structure in the computation

In the previous section we presented the code template which computed the adjoint of a full timestep at once. In this section we describe the concept of "AD of stencil codes." The basic idea is to further expose the stencil structure of the computation to come up with a larger extended function to be differentiated via AD.

First, some notation: in 1-D computations of spatial dimension $n$, there are two end points and $n-2$ center or regular points; the stencil structure is illustrated in Figure 6.9. One time step of forward computation of a 1-D (spatial) code can typically be written as shown in figure 6.10. You have three stencil codes: $f1$ computes the next state of the left end based on previous two states; $f3$ computes the next state of the right end point; and $f2$ computes all the center points. All the finite difference schemes have some kind of stencils associated with them, and in this 1-D example, the stencil is rather simple and is based on only one neighbouring point on either side of the previous state.

The adjoint computation can now be written as shown in Figure 6.11 using the templates developed in Chapter 5.

AD is employed to compute the partials above. It is very cheap to do this since the function f1, f2, and f3 are tiny computations and the AD tool in reverse mode won't have any problem dealing with these stencil codes. It is trivial to employ ADMIT-1 to compute the partials of f1, f2 and f3.

[1] There are techniques which avoid storing each intermediate value, involving a tradeoff between space and time, e.g. Griewank's checkpointing scheme [Gri92b].

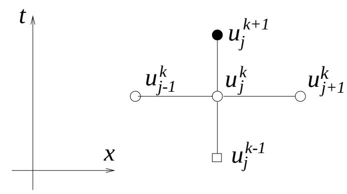



Figure 6.9: The stencil for the 1-D problem for $j \neq 0, n$. Boundary nodes are slightly different and require separate treatment.

```
function unew=timestep(c, ucur, uold)
% left end
unew(1) = f1(c(1), ucur(1), ucur(2), uold(1));
%The center points
for i = 2 : n − 1
    unew(i) = f2(c(i), ucur(i − 1), ucur(i), ucur(i + 1), uold(i));
end
% Right end
unew(n) = f3(c(n), ucur(n − 1), ucur(n), uold(n));
```

Figure 6.10: The 1-D (spatial) stencil code

The complete adjoint code can now be developed using the adjtimestep template. The order of the computation will be reversed: in the stencil adjoint code (Figure 6.11) we compute the adjoint of the right end first, then the center points and finally the left end, and in the full adjoint code we compute adjoints starting from timestep $K$ and go to timestep 1. We skip the code template for the full adjoint; please refer to [CSV] for detailed development of adjoint codes.

### Stencils as Projections

In general, we can represent the stencil operators as projection operators, and we can develop general adjoint schemes using these operators. This will capture more general timestepping schemes, where the stencils can be large (one point depends on many neighbours, etc.) or different standard stencils used in finite difference/finite element schemes. The aim of introducing projection operators is to capture the generality of stencil codes, so that the code templates we present here may be applied to all timestepping codes.

Mathematically, the forward computation using the projection operators can be written as shown in Figure 6.12.

Here $Q_i$ denotes the projection operators, e.g. for the 1-D example presented above $Q_1$ will pick up elements 1 and 2, $Q_i, i = 2 : n-1$ will pick up three elements $i-1, i, i+1$ and $Q_n$ will pick up elements $n-1$ and $n$. $R_i$ always picks up only the element $i$ from uold. Using this notation, it is easy to develop adjoint code as shown in Figure 6.13.

AD can be employed in a trivial manner to compute the partials above. It is trivial to generalize this scheme for complicated 2-D stencil codes, once you figure

**function** $[adjucur, adjuold, adjc]$=**adjtimestep**$(adjunew, c, ucur, uold)$
% Initialize
$adjuold = zeros(n,1); adjucur = zeros(n,1); adjc = zeros(n,1);$
% right end
$adjuold(n) = adjuold(n) + adjunew(n) * \frac{\partial f3}{\partial uold(n)};$
$adjucur(n-1) = adjucur(n-1) + adjunew(n) * \frac{\partial f3}{\partial ucur(n-1)};$
$adjucur(n) = adjucur(n) + adjunew(n) * \frac{\partial f3}{\partial ucur(n)};$
$adjc = adjc + adjunew(n) * \frac{\partial f3}{\partial c(n)};$
%The center points
**for** $i = n-1 : -1 : 2$
$\quad adjuold(i) = adjuold(i) + adjunew(i) * \frac{\partial f2}{\partial uold(i)};$
$\quad adjucur(i-1) = adjucur(i-1) + adjunew(i) * \frac{\partial f2}{\partial ucur(i-1)};$
$\quad adjucur(i) = adjucur(i) + adjunew(i) * \frac{\partial f2}{\partial ucur(i)};$
$\quad adjucur(i+1) = adjucur(i+1) + adjunew(i) * \frac{\partial f2}{\partial ucur(i+1)};$
$\quad adjc = adjc + adjunew(i) * \frac{\partial f2}{\partial c(i)};$
**end**
% left end
$adjuold(1) = adjuold(1) + adjunew(1) * \frac{\partial f1}{\partial uold(1)};$
$adjucur(1) = adjucur(1) + adjunew(1) * \frac{\partial f1}{\partial ucur(1)};$
$adjucur(2) = adjucur(2) + adjunew(1) * \frac{\partial f1}{\partial ucur(2)};$
$adjc = adjc + adjunew(1) * \frac{\partial f1}{\partial c(1)};$

Figure 6.11: The adjoint stencil code

out the number of stencil codes needed (in this case three, f1, f2 and f3) and the projection operators for your finite difference scheme.

## 6.5 Option Pricing Problem

Now we consider an application in finance, that of option pricing. To begin with we explain the basics of option pricing and the market parameters involved. For detailed information about option pricing we urge you to refer to Jarrow and Turnbull [JT96] and Hull [Hul98].

- **Options:** There are two kinds of options, *calls* and *puts*, and they come in two different flavors, namely *European* and *American*. A European call option gives its owner a right to *buy* a common stock at a fixed price (exercise or strike price $K$) on a fixed future date (time $T$). The European put options gives a right to *sell* a common stock at a fixed price on a fixed future date (time $T$). The value of the option is denoted by $V$ which can be a function of the current stock price and time $V(S,t)$. Typically we know the value of an option at maturity date $t = T$ E.g., for European put the option is worthless if at time $T$ the stock price $S(T) > K$; otherwise it has value $K - S(T)$; this can be written as $V(S(T),T) = \max(K - S(T), 0)$. In order to judge the worth of an option we want to compute its value at the current time, i.e. at time $t = 0$; this involves the simulation of the options market which is usually done using

**function** $unew$=**timestep**$(c, ucur, uold)$
% left end
$unew(1) = f1(P_1(c), Q_1(ucur), R_1(uold));$
%The center points
**for** $i = 2 : n - 1$
    $unew(i) = f2(P_i(c), Q_i(ucur), R_i(uold));$
**end**
% Right end
$unew(n) = f3(P_n(c), Q_n(ucur), R_n(uold));$

Figure 6.12: The 1-D (spatial) stencil code using projection operator

**function** $[adjucur, adjuold, adjc]$=**adjtimestep**$(adjunew, c, ucur, uold)$
% Initialize
$adjuold = zeros(n, 1); adjucur = zeros(n, 1); adjc = zeros(n, 1);$
% right end
$P_n(adjc) = P_n(adjc) + adjunew(n) * \frac{\partial f3}{\partial P_n(c)}$
$R_n(adjuold) = R_n(adjuold) + adjunew(n) * \frac{\partial f3}{\partial R_n(uold)}$
$Q_n(adjucur) = Q_n(adjucur) + adjunew(n) * \frac{\partial f3}{\partial Q_n(ucur)}$
%The center points
**for** $i = n - 1 : -1 : 2$
    $P_i(adjc) = P_i(adjc) + adjunew(i) * \frac{\partial f2}{\partial P_i(c)};$
    $R_i(adjuold) = R_i(adjuold) + adjunew(i) * \frac{\partial f2}{\partial R_i(uold)};$
    $Q_i(adjucur) = Q_i(adjucur) + adjunew(i) * \frac{\partial f2}{\partial Q_i(ucur)};$
**end**
% left end
$P_1(adjc) = P_1(adjc) + adjunew(1) * \frac{\partial f2}{\partial P_1(c)}$
$R_1(adjuold) = R_1(adjuold) + adjunew(1) * \frac{\partial f1}{\partial R_1(uold)}$
$Q_1(adjucur) = Q_1(adjucur) + adjunew(1) * \frac{\partial f1}{\partial Q_1(ucur)}$

Figure 6.13: The adjoint stencil code using projection operators

the Black-Scholes model. *American options* are different from the European ones in that they can be exercised at anytime prior to the maturity date $T$.

- **Market parameters:** The option market depends on te market parameters such as:

    1. *Interest rate:* This is denoted by $r$.
    2. *Volatility:* This is denoted by $\sigma$.
    3. *Dividend rate:* This is denoted by $q$.
    4. *Option parameters:* These include the stock price $S$, the exercise price $K$ and the maturity time $T$.

The pricing of options is done using the Black-Scholes model. We consider the solution of an *American Put* problem here. The Black-Scholes PDE is given by equation (6.12).

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + (r-q)S\frac{\partial V}{\partial S} - rV = 0. \tag{6.12}$$

One way to solve the Black-Scholes PDE is to use an implicit finite difference scheme running backwards ($k = N-1:1$) as shown in equation (6.13).

$$\frac{V_j^{k+1} - V_j^k}{\Delta t} + \frac{1}{2}\sigma^2 S_j^2 \frac{V_{j+1}^k - 2V_j^k + V_{j-1}^k}{\Delta h^2} + (r-q)S_j\frac{V_{j+1}^k - V_j^k}{\Delta h} - rV_j^k = 0. \tag{6.13}$$

We can write the implicit scheme in a matrix vector notation as:

$$A(\sigma, r, q)V^k = V^{k+1},$$

where, matrix $A$, it turns out, is a tridiagonal matrix.

The boundary conditions for the American put problem are:

$$V(S,T) = \max(K - S, 0), \tag{6.14}$$
$$V(0,t) = K, \tag{6.15}$$
$$V(Smax, t) = 0. \tag{6.16}$$

A solution of an American put problem is shown in the Figure 6.14, where the solution at maturity $t = T$ is given by the boundary conditions above.

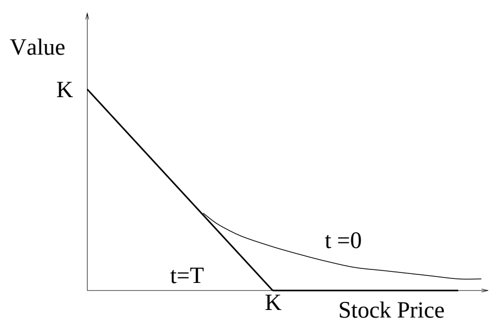



Figure 6.14: American put solution at Maturity $t = T$ and at $t = 0$

The "structured" computation of $V(S,T)$ can be written as shown in Figure 6.15.

Differentiating, the extended Jacobian is given by:

$$J_E = \left(\begin{array}{c|cccccc} A_\sigma V^{N-1} & A & & & & \\ A_\sigma V^{N-2} & -I & A & & & \\ \vdots & & \ddots & \ddots & & \\ A_\sigma V^2 & & & -I & A & \\ A_\sigma V^1 & & & & -I & A \\ \hline & & & & & I \end{array}\right)$$

$$\begin{array}{l}
\textit{Solve for } V^{N-1}: \ A(\sigma, r, q)V^{N-1} = V^N; \\
\textit{Solve for } V^{N-2}: \ A(\sigma, r, q)V^{N-2} = V^{N-1}; \\
\vdots \\
\textit{Solve for } V^1: \ A(\sigma, r, q)V^1 = V^2; \\
\textit{Compute } F := V^1 - \phi_{tar}.
\end{array}$$

Figure 6.15: Structured program for American Put options

Given this setting of a structured problem, there might a variety of problems relating to computing or estimating market parameters given some market data corresponding to a single option or a portfolio of options. The parameters which are usually estimated are $\sigma, r, q$ and these are computed using a nonlinear least-squares solution, which will require derivatives of the structured program w.r.t. $\sigma, r, q$.

# Chapter 7

# Design of Structured Algorithms for Optimization

In this chapter we present the concept of "structured algorithms" for optimization; in particular we present some examples of incorporating problem structure in the optimization algorithm design. The material presented here can be taken as "food for thought" for the optimization algorithm design community and will help optimizers to "think structure" while devising new algorithms.

First we describe the benefits structure exploitation provides with regards to computing with implicit computations (a simple example of an implicit computation being the the solution of the nonlinear equations $F(x, y) = 0$, where $y(x)$ is defined via $F$).

We also present a new preconditioning strategy for use in a PCG algorithm to solve large-scale minimization problems with linear constraints. The work developed here is motivated via preconditioning of extended Hessian matrices, which are used for efficient solution of Newton steps for structured minimization problems. The preconditioning strategy can be generalised both to nonlinear constraints and inequality constraints without loss of generality. The complete details of the preconditioning strategy can be found in [CVb].

In Section 7.3 we present recipes to compute information related to original Jacobian and Hessian matrices from extended Jacobian and Hessian matrices. In Chapter 3, we have already seen how to compute Newton steps using extended Jacobian and Hessian matrices without computing true Jacobian or Hessian matrices. In Section 3 we list other computations we can do without forming the true Jacobian or Hessian matrix.

## 7.1 Implicit Computations

### 7.1.1 Nonlinear equations

In this section we consider the computation $F(x, y) = 0$, yielding an implicit definition of $y \in \Re^m$ uniquely given $x \in \Re^n$. Let us represent the corresponding computation which defines $y$ directly as $y = G(x)$. An important observation is that $F$ and $G$ have different complexities, typically $F$ is cheap but $G$ is expensive. For example suppose $F \equiv Ay - x$ then $G \equiv A\backslash x$, so $F$ involves just the matrix vector product and $G$ involves the solution.

We are interested in computing the Jacobian matrix $J = \frac{dy}{dx}$. The straightforward way to compute $J$ is to let AD differentiate through the whole function evaluation of $G$. Considering that $J$ will be dense (which is typically the case in implicit computations, unless the computation is trivial, e.g., explicit when $F$ and $G$ will be same), the time complexity will be proportional to $\min(m, n)$ times the time to compute $y$ itself or $\min(m, n)$ evaluations of $G$. But $G$ can be a very complicated function, e.g., it might involve iterations to compute $y$ given $x$, hence the above complexity can be impractical.

However, the derivative can be computed outside of the iterative process and we can get rid of $\min(m, n)$ evaluations of $G$ by considering the following implicit AD scheme. Differentiating equation $F(x, y) = 0$ w.r.t. $x$, we get:

$$F_x(x, y) + F_y(x, y)\frac{dy}{dx} = 0.$$

Hence computing $J$ just involves a linear system solve:

$$\frac{dy}{dx} = -F_y(x, y)^{-1}F_x(x, y). \tag{7.1}$$

This method in equation (7.1) does not involve differentiating through the computation of function $G$, but just the implicit definition $F$ with respect to both $x$ and $y$ parameters, and typically $F_x, F_y$ are sparse Jacobians. Computation of $y$, i.e., function $G$ needs to be done only once, and the derivatives which need to be computed are sparse and can be computed cheaply since $F$ is cheap.

The forward product can be computed,

$$JV = -F_y(x, y)^{-1}(F_x(x, y)V);$$

similarly the adjoint product can be computed,

$$J^TW = -F_x(x, y)^T(F_y(x, y)^{-T}W).$$

To compute the second order derivatives, differentiate $F(x, y) = 0$ twice:

$$F_{xx}(x, y) + F_{xy}^T\frac{dy}{dx} + (\frac{dy}{dx})^T F_{xy} + (\frac{dy}{dx})^T F_{yy}\frac{dy}{dx} + F_y\frac{d^2y}{dx^2} = 0.$$

Solving for $\frac{d^2y}{dx^2}$ will involve a solution in system $F_y$.

### 7.1.2 Parametric optimization

The parametric optimization problem is another example of an implicit computation. In this section we'll illustrate how to exploit structure in a parametric optimization problem.

- **The 1-D problem** Consider the optimization problem:

$$\min_t \quad g(x^*(t)), \quad t \in \Re^1, g \in \Re^n \to \Re^1 \tag{7.2}$$

$x^*(t)$ is defined by: min $\{f(x) : Ax = b(t)\}$, $A \in \Re^{m \times n}$, $x \in \Re^n$, $b \in \Re^m$.

The optimal solution $x^*(t)$, for a given $t$ satisfies the nonlinear system of equations:

$$\nabla f(x^*) + A^T \lambda^* = 0 \tag{7.3}$$
$$Ax^* = b \tag{7.4}$$

For problem (7.2), we need the first and second order derivatives $g'$ and $g''$:

$$\frac{dg}{dt} = \nabla_x g(x^*)^T \frac{dx^*}{dt} \tag{7.5}$$
$$\frac{d^2 g}{dt^2} = \frac{dx^*}{dt}^T \nabla_{xx} g(x^*) \frac{dx^*}{dt} + \nabla_x g(x^*)^T \frac{d^2 x^*}{dt^2} \tag{7.6}$$

For computing these two quantities we need to compute the sensitivities of $x^*$, i.e. $\frac{dx^*}{dt} \in \Re^n$ and $\frac{d^2x^*}{dt^2} \in \Re^n$.

  - **How to compute $\frac{dx^*}{dt}$**
    Differentiate the equations (7.3) and (7.4) w.r.t. $t$:

$$\nabla^2 f(x^*) \frac{dx^*}{dt} + A^T \frac{d\lambda^*}{dt} = 0 \tag{7.7}$$
$$A \frac{dx^*}{dt} = \frac{db}{dt} \tag{7.8}$$

or

$$\begin{pmatrix} \nabla^2 f(x^*) & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} \frac{dx^*}{dt} \\ \frac{d\lambda^*}{dt} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{db}{dt} \end{pmatrix} \tag{7.9}$$

The above system is of size $(m+n) \times (m+n)$, with $\frac{dx^*}{dt} \in \Re^n$, $\frac{d\lambda^*}{dt} \in \Re^m$.

  - **How to compute $\frac{d^2x^*}{dt^2}$**
    Differentiate the equations (7.7) and (7.8) w.r.t. $t$:

$$H_x \frac{dx^*}{dt} \frac{dx^*}{dt} + H \frac{d^2 x^*}{dt^2} + A^T \frac{d^2 \lambda^*}{dt^2} = 0 \tag{7.10}$$
$$A \frac{d^2 x^*}{dt^2} = \frac{d^2 b}{dt^2} \tag{7.11}$$

or

$$\begin{pmatrix} \nabla^2 f(x^*) & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} \frac{d^2x^*}{dt^2} \\ \frac{d^2\lambda^*}{dt^2} \end{pmatrix} = \begin{pmatrix} -H_x \frac{dx^*}{dt}\frac{dx^*}{dt} \\ \frac{d^2b}{dt^2} \end{pmatrix}.$$

The RHS and the solution vector are both $(m+n)$-vectors. $H_x$ is a $n \times n \times n$ tensor, and when multiplied twice by $\frac{dx^*}{dt}$ a $n$-vector, the result $-H_x \frac{dx^*}{dt}\frac{dx^*}{dt}$ is a $n$-vector too.

- **Multi-D with general nonlinear constraints**

  In this case $x^*(t)$ is defined by: $\min \quad \{f(x,t) : c(x,t) = 0\} \quad x \in \Re^n, \quad t \in \Re^s, \quad f \in \Re^{n+s} \to \Re, f \in \Re^{n+s} \to \Re^m, m \leq n$, i.e., we have general nonlinear constraints and the parameter $t$ is vector-valued. The optimal solution $x^*(t)$ for a given $t$ satisfies the nonlinear system of equations (7.12) and (7.13).

$$\nabla_x f(x^*,t) + (\nabla_x c(x^*,t))^T \lambda^* = 0, \tag{7.12}$$

$$c(x^*,t) = 0. \tag{7.13}$$

  - **How to compute $\frac{\partial x^*}{\partial t}$**

    Define the Lagrange function: $L(x,t,\lambda) \equiv f(x,t) + c(x^*,t)' * \lambda$. The equations (7.12) and (7.13) are equivalent to:

$$\nabla_x L = 0 \tag{7.14}$$

$$c(x^*,t) = 0. \tag{7.15}$$

    Differentiating (7.14 – 7.15):

$$\nabla^2_{xx} L \frac{\partial x^*}{\partial t} + \nabla_{xt} f + (\nabla_{xt} c)^T \lambda^* + (\nabla_x c(x^*,t))^T \frac{\partial \lambda^*}{\partial t} = 0 \tag{7.16}$$

$$(\nabla_x c(x^*,t)) \frac{\partial x^*}{\partial t} + \nabla_t c(x^*,t) = 0 \tag{7.17}$$

    or

$$\begin{pmatrix} \nabla^2_{xx} L & (\nabla_x c(x^*,t))^T \\ \nabla_x c(x^*,t) & 0 \end{pmatrix} \begin{pmatrix} \frac{\partial x^*}{\partial t} \\ \frac{\partial \lambda^*}{\partial t} \end{pmatrix} = \begin{pmatrix} -(\nabla_{xt} f + (\nabla_{xt} c)^T \lambda^*) \\ -\nabla_t c(x^*,t) \end{pmatrix}. \tag{7.18}$$

  - **How to compute $\frac{\partial^2 x^*}{\partial t^2}$** Differentiate the equations (7.16) and (7.17) w.r.t. $t$.

    In this case the right hand side and the solution vector will be tensors of size $(m+n) \times t \times t$. One way to solve is to solve $t$ matrix equations separately, solving a tensor panel of size $(m+n) \times t$.

*solve for* $y_1 = (x^*, \lambda^*)$*:*

$$F_1(t, y_1) = \begin{pmatrix} \nabla f(x^*) + A^T \lambda^* \\ A(x^*) - b(t) \end{pmatrix} = 0$$

*output:* $z = x^*$

Figure 7.1: Modified extended function of parametric optimization problem

### Parametric optimization and structure

The parametric optimization solution $x^*$ satisfies the necessary conditions expressed in equations (7.3) and (7.4). Thus locally the solution of the parametric optimization problem can be expressed in terms of the structured function shown in Figure 7.1.

Differentiating the modified extended function shown in Figure 7.1, w.r.t. the independent variable $t$ as well as the intermediates $\lambda^*, x^*$ yields the extended Newton system:

$$\begin{pmatrix} 0 & \nabla^2 f(x^*) & A^T \\ b_t & A & 0 \\ 0 & I & 0 \end{pmatrix} \begin{pmatrix} \delta t \\ \delta x \\ \delta \lambda \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -x^* \end{pmatrix}$$

Note the similarity of this system to the sensitivity system shown in equation (7.9).

### Numerical results

We experimented with the Brown function, which is mathematically given by:

$$y = \sum_{i=1}^{n-1} (x_i^2)^{x_{i+1}^2+1} + (x_{i+1}^2)^{x_i^2+1}.$$

The constraints were of form $Ax = b$, where $A$ has a single row, and $b$ a scalar is the parameter. Table 7.1 shows the result for various sizes of $n$. It shows that the time for a parametric optimization solution is less than the function evaluation of $x^*$ itself on this problem. The parametric optimization solve just consists of the solution of a linear system after $x^*$ has been computed.

## 7.2 Preconditioning Constrained Minimization Problems

The motivation for a new approach to the preconditioning of constrained minimization problems came from structured problems. We have seen that any computation $z = f(x)$ can be written in a structured form as follows :

Table 7.1: Comparisons of parametric optimization function computation to the derivative solve

| size | One function evaluation (of $x^*$) | Parametric optimization solve |
|---|---|---|
| 500 | 4.93 | 0.61 |
| 1000 | 9.07 | 2.13 |
| 2000 | 31.63 | 7.61 |
| 4000 | 91.41 | 23.67 |

*Solve for* $y$ : $F(x, y) = 0$
*"Solve" for output* $z$ : $z \leftarrow g(x, y)$

Figure 7.2: A two step structured program

That is, first intermediate variables $y$ are solved for using $x$, via solving the nonlinear equation $F(x, y) = 0$, and then the output $z$ is formed using $x$ and $y$.

So the problem of minimizing $z = f(x)$ can be alternatively expressed as nonlinear equality constrained minimization problem in both variables $x$ and $y$, i.e.,

$$\min g(x, y), \quad F(x, y) = 0. \tag{7.19}$$

The Newton step for the structured program in Figure 7.2 is given by the Extended Newton system shown in equation (7.20).

$$\begin{pmatrix} \nabla^2_{xx} g & \nabla_{xy} g & F_x^T \\ \nabla^2_{yx} g & \nabla_{yy} g & F_y^T \\ F_x & F_y & 0 \end{pmatrix} \begin{pmatrix} \delta x \\ \delta y \\ \delta \lambda \end{pmatrix} = \begin{pmatrix} -\nabla f \\ 0 \\ 0 \end{pmatrix}. \tag{7.20}$$

However, the KKT system for the minimization problem (equation (7.19)) is given by:

$$\begin{pmatrix} \nabla^2_{xx} g & \nabla_{xy} g & F_x^T \\ \nabla^2_{yx} g & \nabla_{yy} g & F_y^T \\ F_x & F_y & 0 \end{pmatrix} \begin{pmatrix} \delta x \\ \delta y \\ \delta \lambda \end{pmatrix} = \begin{pmatrix} -\nabla_x g \\ -\nabla_y g \\ 0 \end{pmatrix}. \tag{7.21}$$

The two systems (equations (7.20) and (7.20)) look similar (with a difference in the right hand side) and that is exactly what motivates the preconditioner solution. We first look at the linearly constrained problems. First consider the quadratic minimization problem with linear equality constraints

$$\min c^T x + \tfrac{1}{2} x^T H x, \quad Ax = 0.$$

If sparsity is ignored the most straightforward CG-approach is to solve system

$$(Z^T H Z) w = -r$$

where $r$ is the current residual $Z^T(Hx+c)$. $Z$ forms the nullspace of matrix $A$. Inn other words solve the reduced system

$$\bar{H}w = -\bar{c}, \quad x \leftarrow x + Zw$$

where $\bar{H} = Z^T HZ, \bar{c} = Z^T c$.

The preconditioned conjugate-gradient algorithm is shown in Figure 7.3.

PCG

While $||z||$ not small enough
1. $d \leftarrow z + \beta d_-, \qquad \gamma \leftarrow d^T \bar{H} d$
2. If $\gamma \leq 0$ return (p,d)
   else $\alpha \leftarrow \frac{r^T z}{\gamma}, \quad p \leftarrow p + \alpha d, \quad r \leftarrow r - \alpha \bar{H} d$
3. $z \leftarrow \mathcal{P}_k(r)$
4. $\beta \leftarrow \frac{r^T z}{r_-^T z_-}$

Figure 7.3: PCG algorithm

The notation $v_-$, where $v$ is a vector, refers to the value of $v$ in the previous PCG iteration.

The main problem is how to efficiently implement the preconditioning step, $z \leftarrow \mathcal{P}_k(r)$, i.e., approximate the system

$$(Z^T HZ)w = -r. \tag{7.22}$$

Usual preconditioning strategy do not work because $Z^T HZ$ is not sparse enough, however, in general for large-scale problems $H$ is typically very sparse. This problem was also treated in [Col94]. Here we present a completely general approach leading to a general class of preconditioners.

## A general approach

The structured Newton step approach discussed in [CV96b,CV96a] yields a general preconditioner idea – consider the linear system:

$$G(w) \equiv (Z^T HZ)w + Z^T c = 0.$$

The "structured" program shown in Figure 7.4 evaluates $G(w)$.

The Newton step via the extended Jacobian matrix is given by

$$J_E = \begin{pmatrix} Z & -I & \\ & H & -I \\ & & Z^T \end{pmatrix} \begin{pmatrix} w \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -G(w) = -r = -(Z^T HZw + Z^T c) \end{pmatrix} \tag{7.23}$$

| |
|---|
| *Solve for* $y_1 : y_1 := Zw$ $\quad$ (*i.e.*, $Zw - y_1 = 0$) |
| *Solve for* $y_2 : y_2 := Hy_1$ $\quad$ (*i.e.*, $Hy_1 - y_2 = 0$) |
| *Solve for* $G(w) = Z^T y_2 + Z^T c$ |

Figure 7.4: Program to compute $F(w)$

Therefore, an approximate Newton step is given by

$$J_E = \begin{pmatrix} \tilde{Z} & -I & \\ & \tilde{H} & -I \\ & & \tilde{Z^T} \end{pmatrix} \begin{pmatrix} z \\ v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ -r = -Z^T(Hx+c) \end{pmatrix}$$

where $x \equiv Zw$ and $\tilde{Z}$ and $\tilde{H}$ are (sparse) approximations to $Z$ and $H$ respectively. This is equivalent in theory to the system

$$\tilde{Z}^T \tilde{H} \tilde{Z} z = -r. \tag{7.24}$$

Hence equation (7.24) appears to represent a possible preconditioning strategy provided $\tilde{H} > 0$. Sufficiently sparsity in $\tilde{Z}$ and $\tilde{H}$ will allow for an efficient solution to equation (7.24).

A difficulty with this general approach is that it necessary to compute $\tilde{Z}$ explicitly; equation (7.24) requires $\tilde{Z}$. $Z$ is needed only in a implicit form to enable formation of products of form $Zv$ and $Z^T v$; note that the product $r^T z = (Hd+c)^T Zz$ requires multiplying a vector by $Z$ .

In the following sections, we build on this general approach to come up with a preconditioning strategy which doesn't require forming any of $\tilde{Z}$ or $Z$ .

## Using A fundamental basis

The precondition system equation (7.22):

$$(Z^T H Z)w = -r = -Z^T(Hx+c) \tag{7.25}$$

can also be written as

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} w' \\ u \end{pmatrix} = \begin{pmatrix} -(Hx+c) \\ 0 \end{pmatrix} \tag{7.26}$$

where $w' = Zw$.

Earlier, preconditioners for this problem only involved approximating $H$ by a suitable positive definite approximation $\tilde{H}$ [Col94], i.e.

$$\begin{pmatrix} \tilde{H} & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} w' \\ u \end{pmatrix} = \begin{pmatrix} -(Hx+c) \\ 0 \end{pmatrix}. \tag{7.27}$$

Here we present an improvement on the above idea, where it is also possible to incorporate an approximation of $A$ into the system. This scheme is described in the rest of this section.

The fundamental null basis is given by equation (7.28). We assume that $A = [A_1 \quad A_2]$, where $A_1$ is square and nonsingular.

$$Z = \begin{pmatrix} -A_1^{-1}A_2 \\ I \end{pmatrix}. \tag{7.28}$$

It can be shown that the system (7.26) is equivalent to the system (7.29), in terms of the solution vector $w$ [CVb].

$$\begin{pmatrix} H & A^T \\ A & 0 \end{pmatrix} \begin{pmatrix} w' \\ \hat{u} \end{pmatrix} = \begin{pmatrix} 0 \\ -Z^T(Hx+c) \\ 0 \end{pmatrix} \tag{7.29}$$

Therefore the precondition step can be computed via

$$\begin{pmatrix} \tilde{H} & \tilde{A^T} \\ \tilde{A} & 0 \end{pmatrix} \begin{pmatrix} z' \\ \tilde{u} \end{pmatrix} = \begin{pmatrix} 0 \\ -Z^T(Hx+c) = -r \\ 0 \end{pmatrix} \tag{7.30}$$

$$(\tilde{Z}^T\tilde{H}\tilde{Z})z = -r, z' = \tilde{Z}z$$

and let $x = Zz$, where $\tilde{Z}$ is obtained from $\tilde{A}$ as is $Z$ from A in (7.28).

Also it is not strictly necessary to compute $\tilde{Z}$: if $z'$ is written as

$$z' = \begin{pmatrix} z_1' \\ z_2' \end{pmatrix}$$

then $z' = \tilde{Z}z$ along with the form of the fundamental basis in equation (7.28) tells us that $z = z_2'$; hence the product $r^Tz = r^Tz_2'$ and $z_2$ is available from the solution of system (7.30). Also, if you need the current iterate $x = Zz' = Zz_2'$, you don't need direct access to $\tilde{Z}$.

It can be shown that this fundamental basis approach is a special case of the general approach outlined in §3. For this, we show that the system (7.23) is equivalent to (7.29); we again refer the reader to [CVb] .

## Algorithm

In this section we describe a scheme to drop nonzeros from $A$, based on a tolerance scheme, with $\tilde{A}$ a very close approximation to $A$ for low tolerance values. In the scheme described below the values in $A$ below the tolerance in a relative sense are dropped. The scheme consists of two major steps.

- **Normalization :** In this step, all rows of $A$ are normalized to have unit norm. This is important, because for removing certain "small" nonzeros, we need to quantify what "small" is, and normalizing each constraint gives us a good idea of that.
- **Removal of "small" nonzeros :**

  This step actually removes certain nonzeros which are considered small enough, based on the input parameter, *tol*. In particular, a nonzero which is smaller in absolute value than *tol* times the largest (in absolute value) entry in that column, is dropped.

The M-file which implements the scheme above is shown in Figure 7.5.

```
function A = gangstr(M,tol)
% Implement approximation of A by removing some nonzeros
   [m,n] = size(M);
   if nargin < 2, tol = 1e-2; end
% Normalize M
   Msqr=M.*M;
   X=sum(Msqr');
   X(find(X==0))=ones(length(find(X==0)),1);
   X=X(:);
   M=spdiags(1./sqrt(X),0,m,m)*M;
% Remove nonzeros
   dim = sprank(M); sprA = 0; A=M;
   [I,J,V]=find(M);
   while sprA < dim
         absM=abs(M);
         maxvec=full(tol*max(absM));
         maxvec=maxvec(:); tobekept=find(abs(V) > maxvec(J));
         A=sparse(I(tobekept),J(tobekept),V(tobekept),m,n);
         sprA = sprank(A);
         tol = tol/10;
   end
```

Figure 7.5: Matlab Code to compute $\tilde{A}$ given $A$

## Computational Results

Consider the following sample problem :

$$\min \nabla c^T x + \tfrac{1}{2} x^T H x$$
$$\text{subject to } Ax = 0$$

with $f : \Re^n \to \Re$ and $A \in \Re^{m \times n}$ and $c$ and $H$ were chosen to be the gradient and the Hessian matrix for the 'Brown" function at a given point, which is given by :

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

This function has a tridiagonal Hessian matrix.

Figure 7.6 shows a sample result of applying the preconditioner for $n = 900$. For the preconditioner, $\tilde{H}$ was chosen as a diagonal approximation to $H$ and $A$ was chosen as a sample linear equality test matrix of size $358 \times 900$. $\tilde{A}$ was formed based on the scheme described before with varying tolerance.

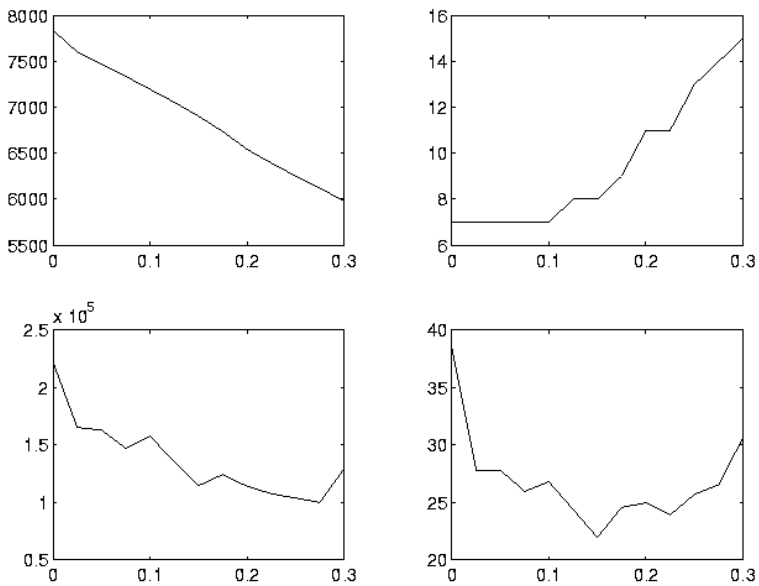



Figure 7.6: A sample preconditioner result

The $x$-axis in all 4 plots denotes the tolerance, *tol*, varying from 0 to 0.3. Plot (1,1) plots, the number of nonzeros in the preconditioner as a function of *tol*. Plot (2,1) plots the combined sum of nonzeros in the LU factors of the preconditioner, as a function of *tol*. Plot (1,2) plots the number of CG iterations and finally plot (2,2) plots the actual amount of time taken (in second) to solve the system using CG.

As you see in Figure 7.6, the number of nonzeros in the preconditioner drop as the tolerance is increased, and so does the number of nonzeros in the LU factors. In practice, however, the nonzeros in LU factorization don't necessarily go down as nonzeros in the preconditioner go down, but statistically, the decrease is generally observed. The number of iterations, as expected, is a monotonically increasing function of the tolerance, since as the tolerance is increased, more elements are dropped from $A$ and $\tilde{A}$ becomes a worse approximation to $A$. Actually, the number of iterations behaves more like a staircase function. Initially for small tolerances, $\tilde{A}$ is still pretty close to $A$, so the number of iterations remain the same. The number of iterations go up when $\tilde{A}$ becomes sufficiently different from $A$. The most interesting plot is the time-plot, which shows that the time required for the CG process as a function of tolerance. Initially, as the elements are dropped, the time per iteration decreases (LU factors are sparser) and the number of iterations remains constant, so the overall time decreases. But finally, when the number of iterations becomes too large, the total time also increases. Hence there is a tolerance *tol=optim*, for which the time required is optimal.

## 7.3 Computing with Extended Derivative Matrices

Let us look at the block matrix representation of the extended Jacobian matrix:

$$J_E = \left( \begin{array}{c|c} A & L \\ \hline B & M \end{array} \right). \tag{7.31}$$

Similarly the extended Hessian Matrix in symmetric form can be written in the form (recall equation 3.14):

$$H_E^S = \begin{pmatrix} S & M \\ M^T & B \end{pmatrix}. \tag{7.32}$$

We have seen that the Newton step and the original Jacobian/Hessian can be recovered from the extended Jacobian/Hessian matrix. Here we list the computations we can do without forming the original Jacobian or Hessian.

- **Jacobian-related computations**
  - $(J + C)s = f$
    Can be solved by solving the linear system:
    $$\left( \begin{array}{c|c} A & L \\ \hline B+C & M \end{array} \right) \begin{pmatrix} s \\ temp \end{pmatrix} = \begin{pmatrix} 0 \\ f \end{pmatrix}.$$
  - $CJs = f$
    This "composite" system can be solved by solving the linear system:
    $$\begin{pmatrix} A & L & 0 \\ B & M & -I \\ 0 & 0 & C \end{pmatrix} \begin{pmatrix} s \\ temp_1 \\ temp_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ f \end{pmatrix}.$$
  - $J^T Js = f$
    This kind of system arises in the least squares solution. We solve it via
    $$\begin{pmatrix} -I & 0 & M & B \\ 0 & 0 & L & A \\ M^T & L^T & 0 & 0 \\ B^T & A^T & 0 & 0 \end{pmatrix} \begin{pmatrix} v_2 \\ v_3 \\ v_1 \\ s \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ f \end{pmatrix}$$

– $det(J)$ We can also solve for the determinant of the true Jacobian using the extended Jacobian using the following scalar equation.

$$det(J) \cdot det(L) = det \begin{pmatrix} L & A \\ M & B \end{pmatrix}$$

This recipe is derived by taking determinants of both sides of the following identity:

$$\begin{pmatrix} L & A \\ M & B \end{pmatrix} = \begin{pmatrix} L & 0 \\ M & J \end{pmatrix} \begin{pmatrix} I & L^{-1}A \\ 0 & I \end{pmatrix}$$

– $LU(J)$
The $LU$ factorization of the true Jacobian matrix can be based on extended Jacobian matrix. Assume the following $LU$ factorization of the permuted extended Jacobian matrix:

$$\begin{pmatrix} L & A \\ M & B \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} U_{11} & U_{12} \\ 0 & U_{22} \end{pmatrix}.$$

Then

$$J = L_{22}U_{22}.$$

– $eig(J)$
The problem of calculating the eigenvalues of $J$ can be transformed into a generalised eigenvalue problem on the components of $J_E$. Firstly, without loss of generality assume that $B \equiv 0$, since if it is not, then an additional intermediate variable can be introduced, so that the final variable $z$ doesn't depend on $x$.
Hence $J = -ML^{-1}A$, so the eigenvalue problem is $-ML^{-1}Ax = \lambda x$. Let's define $z = Ax$ and $w = L^{-1}z$ so that:

$$-Mw = \lambda x.$$

Premultiply by $A$:

$$-AMw = \lambda Ax = \lambda z = \lambda Lw$$

Hence

$$-AMw = \lambda Lw$$

a generalised eigenvalue problem. The dimension of this eigenvalue problem is higher than that of the eigenvalue problem for the true Jacobian matrix, so we will have more eigenvalues, but it can be shown that the extra ones will all be zeros, because $A * M$ will be a singular matrix of rank at most $n$, but of size $p \times p$, hence at least $p - n$ eigenvalues will be zeros.

If only extreme eigenvalues are sought for, then typically we can directly use power iteration in $J$, since we know a method to compute $Jv$. Inverse power iterations and shifts can also be performed using the first technique in the section.

- **Bound on** $cond(H)$
  Let's look at

$$-AMw = \lambda Lw$$

  Premultiply by $w^T$, it gives us

$$\lambda = \frac{w^T AMw}{w^T Lw}$$

  We get

$$\lambda_{min} \geq \frac{\lambda_{min}(AM)}{\lambda_{max}(L)}$$
$$\lambda_{max} \leq \frac{\lambda_{max}(AM)}{\lambda_{min}(L)}$$

  Hence:

$$\kappa(H) \leq \frac{\lambda_{max}(L)}{\lambda_{min}(AM)} \cdot \frac{\lambda_{max}(L)}{\lambda_{min}(L)} = \kappa(AM)\kappa(L) \leq \kappa(A)\kappa(M)\kappa(L)$$

- $null(J)$
  If $Z = null(J_E)$ then $Z(1:n,:)$ is $null(J)$.

- **Hessian-related Computations**

  A large number of Hessian-related computations can be carried out in the same fashion as the Jacobian related computations, owing to the same lower Hessenberg form of the extended derivative matrices, e.g., $HV$, $(H+C)s = f$, $det(H)$, $LDL^T$ factorization, $rank(H)$, $null(H)$.

  An important thing to note in Hessian computation is the involvement of sparse symmetric solutions in the computation.

  - $V^T HV$
    Using the notation in equation (7.32). Compute $X = M^T V$ and then solve $V^T HV = V^T BV - X^T(S^{-1}X)$ using cholesky factorization of $S$.
  - $Z^T HZs = f$
    Solve

$$\begin{pmatrix} & & -I & Z \\ & S & M & \\ -I & M^T & B & \\ Z^T & & & \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ s \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ f \end{pmatrix}$$

This can be generalised to solve systems of form $(D_1 H D_1 + D_2)s = f$, which arise in box constrained optimization.

– $eig(H)$

By an analysis similar to that of the extended Jacobian:

$$-M^T M w = \lambda S w \tag{7.33}$$

– **Bound on** $cond(H)$

A simple analysis of equation (7.33) gives us the bounds:

$$\kappa(H) \leq \frac{\sigma^2_{max}(M)}{\lambda_{min}(S)} \cdot \frac{\lambda_{max}(S)}{\sigma^2_{min}(M)} = \kappa(M)^2 \kappa(S)$$

It is also possible to do compute $QR$ factorizations and quasi-Newton updates using the extended derivative matrices.

## 7.4 AD and Optimization Software Design

Here we present a tip on "optimization software design." As we saw in Chapter 1, most optimization algorithms involve iterative processes which require some derivative information at every iteration. The new generation optimization software will use automatic differentiation as the derivative computing engine. The design of the optimization software has to be adapted to take into account the transparent use of AD technology. By this we mean that the optimization software would have the same design whether we use AD generated derivatives or the user provided hand-coded derivatives. Here we present an idea which employs object oriented technology for easy use of AD technology beneath the optimization software layer.

### The feval Interface

The function feval is overloaded so that it can return the function value as well as the derivatives by the regular execution and the AD execution respectively depending on how it is invoked. The user needs to present just the function in y=fun(x) form and it can overloaded to look like [f,J]=fun(x) to the optimization software making the plugin of AD technology in optimization software really simple.

This section provides two illustrated examples of the usage of the overload feval interface – once each of a vector mapping and a scalar mapping; examplefun.m and examplesfun.m denote the user functions.

- **Vector Mapping:**

```
>> myfun=ADfun('examplefun');
>>
>> x = ones(10,1);
>> y=feval(myfun,x); <- Compute function value
```

```
>> [f,J]=feval(myfun,x); <- Compute the sparse Jacobian
>>
>> options=setopt('forwprod',ones(10,1)); <- compute  forward product
>> [f,JV]=feval(myfun,x,[],options);
>>
>> options=setopt('revprod',ones(10,1)); <- compute reverse product
>> [f,WJ]=feval(myfun,x,[],options);
>>
>> options=setopt('jacsp');
>> SPJ=feval(myfun,x,[],options); <- compute sparsity pattern
```

- **Scalar Mapping:**

```
>> mysfun=ADfun('examplesfun',1);  <- a scalar problem
>>
>> x = ones(10,1);
>> [v,grad,H]=feval(mysfun,x); <- Compute gradient and sparse Hessian
>>
>> options=setopt('htimesv',eye(10,2)); <- Compute Hessian matrix product
>> HV=feval(mysfun,x,[],options);
>>
>> options=setopt('hesssp'); <- Compute sparse Hessian
>> SPH=feval(mysfun,x,[],options);
```

Here is a description of the the class ADfun, which uses feval as an overloaded method to directly call AD technology to compute derivatives from the standard feval interface.

## ADfun

### Purpose

Prepares a matlab function for automatic derivative computation using ADMAT.

### Synopsis

derivfun=ADfun(funstr)

derivfun=ADfun(funstr,scalar)

### Description

derivfun=ADfun(funstr)

e.g myfun=ADfun('examplefun'), and then use myfun in all overloaded feval calls. Default scalar = 0, i.e. the function is treated as a vector mapping.

derivfun=ADfun(funstr,scalar)

scalar =0  function funstr is a Vector Mapping.

scalar =1 function funstr is a scalar Mapping.

For more information on how "feval" is overloaded and used to compute a variety of derivative objects please refer to the ADMAT user guide [CVa].

# Chapter 8

# AD and Parallelism

Automatic differentiation tools have been in circulation for several years. However, as of yet, there is no AD tool which generates parallel derivative code. There are a lot of avenues for using parallel processing for computational differentiation. In this chapter we present a brief summary of work done in order to obtain parallel derivative code.

Conceptually there can be more than one way to obtain parallel derivative code; one of these ways could be to develop a "Parallel AD tool" which given a parallel code for function evaluation returns a parallel code for the derivative evaluation. Another way could employ parallelizing compiler technology to parallelize the sequential derivative code generated from a sequential AD tool.

Apart from the above two techniques to generate parallel derivative code, there has been some work in techniques for parallelization which are not about building parallel AD tools in general, e.g., the stripmining technique, parallelism in reverse mode [Ben96], parallelism of time-dependent processes by Bischof and Wu [BW97]. For a brief introduction to issues in parallel automatic differentiation refer to [Bis91].

Here is brief summary of work that has been done to develop parallel computational differentiation technology:

- **Parallel AD tools**
  - **Source transformation tools.** Paul Hovland's recent thesis "Automatic differentiation of parallel programs" [Hov97] addresses many issues of parallelising the source transformation AD tools. The main emphasis is on ideas for implementation of parallel source-to-source AD tool based on the message passing model. ADIFOR is used a foundation for building AdiFM and AdiMPIF, AD tools for Fortran M and Fortran with a subset of MPI, respectively.
  - **Operator overloaded tools.** An interesting prospect is integration of MultiMatlab and ADMAT to come up with a high-level parallel AD tool for MATLAB. The high-level concepts needed to make such a tool are illustrated in Section 2 in this chapter.

- **Stripmining**. Stripmining is a technique which lends some parallelism to a sequential AD code. The basic idea is to be able to compute the independent derivatives in parallel, e.g., consider the computation of the Jacobian matrix product $JV$ where V is a matrix having $p$ columns. Now, each of the products $JV(:, i), i \in 1 : p$ can be computed independently in parallel.

- **Coarse grained parallelism arising from AD of some processes.**

  - **Parallelism in time-dependent schemes.** Time-dependent process lend themselves to a computational scheme where, while you are computing the next timestep, computation of the derivative of the previous step can be carried out in parallel. Chris Bischof and Po-Ting Wu have analyzed time-dependent processes [BW97].

  - **Structured computation and parallelism.** This scheme which we have developed allows one to exploit generalised structure as outlined in Chapter 3 in the optimization context. We present this scheme in Section 1 in full detail.

In the rest of this chapter we demonstrate two ideas for parallel computational differentiation; the first one is applicable to parallelism in structured computations and the second one is about parallelism in AD of matrix-vector operations.

## 8.1 Parallelism in Structured Computations

We consider the setting of solving the nonlinear equations solution $F(x) = 0$, where you need the Newton step $\delta x = -J^{-1}F$ for an iterative solution. In this section we exploit parallelism in the structured evaluation of $F$ for the computation of the Newton step. First we recollect structured computation from Chapter 3.

### Sequential structured computation

A natural way to evaluate the nonlinear systems $z = F(x)$ is via the lower Hessenberg program illustrated in Figure 8.1 where we assume equation $i$ *uniquely* determines $y^i$, $i = 1 : p$. The program in Figure 8.1 can be differentiated to give the

$$
\begin{array}{l}
\textit{Solve for } y_1 : \ F_1(x, y_1) = 0 \\
\textit{Solve for } y_2 : \ F_2(x, y_1, y_2) = 0 \\
\quad \vdots \\
\textit{Solve for } y_p : \ F_p(x, y_1, y_2, \ldots, y_p) = 0 \\
\textit{``Solve'' for output } z : \ z - F_{p+1}(x, y_1, y_2, \ldots, y_p) \ = \ 0
\end{array}
$$

Figure 8.1: A General Structured Computation

extended Newton system:

$$J_E \begin{pmatrix} \delta x \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ -F \end{pmatrix} \tag{8.1}$$

where

$$J_E = \left(\begin{array}{c|cccc} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial y_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial y_1} & \cdots & & \frac{\partial F_p}{\partial y_p} \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial y_1} & \cdots & & \frac{\partial F_{p+1}}{\partial y_p} \end{array}\right). \tag{8.2}$$

## Parallel structured program

The structure of computation we have dealt with so far is essentially sequential; i.e. for computing $y_i$ you need to first compute $y_{i-1}$. But parallelism can be introduced by the following trick of introducing extra independent variables, and writing the whole computation implicitly as shown in Figure 8.2:

*Solve for* $y_1 :\ F_1(x, y_1) = 0$
*"Satisfy"* $\hat{y}_1 = y_1$
*Solve for* $y_2 :\ F_2(x, \hat{y}_1, y_2) = 0$
*"Satisfy"* $\hat{y}_2 = y_2$
$\vdots$
*Solve for* $y_p :\ F_p(x, \hat{y}_1, \hat{y}_2, \ldots, y_p) = 0$
*"Satisfy"* $\hat{y}_p = y_p$
*"Solve" for output* $z :\ z - F_{p+1}(x, \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_p) \ = \ 0$

Figure 8.2: A General "Parallel" Structured Computation

Here there are additional intermediate variables, namely $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_p$. But the result is that each of the $p$ subcomputations $F_1$ through $F_p$ can be carried off in **parallel**. Note that this "parallel" computation is not the same as the sequential computation in figure 8.1 since the nonlinear function whose zero we are finding and the set of independent variables $x$ are different in these two settings – however the goal is the same: that of finding $x$ which drives $F(x)$ to 0. More specifically, in the parallel version we are finding a zero of a different nonlinear function $F_E$, whose arguments include the intermediate variables aside from the independent variable $x$.

The program in Figure 8.2 can be differentiated to give the extended Newton system:

$$J_E \begin{pmatrix} \delta x \\ \delta \hat{y}_1 \\ \delta \hat{y}_2 \\ \vdots \\ \delta \hat{y}_p \\ \delta y_1 \\ \delta y_2 \\ \vdots \\ \delta y_p \end{pmatrix} = \begin{pmatrix} 0 \\ -(\hat{y}_1 - y_1) \\ 0 \\ -(\hat{y}_2 - y_2) \\ 0 \\ \vdots \\ -F \end{pmatrix} \tag{8.3}$$

where

$$J_E = \left(\begin{array}{cccc|cccc} \frac{\partial F_1}{\partial x} & & & & \frac{\partial F_1}{\partial y_1} & & & \\ & I & & & -I & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial \hat{y}_1} & & & & \frac{\partial F_2}{\partial y_2} & & \\ & & I & & & -I & & \\ \vdots & & & \ddots & & & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial \hat{y}_1} & \cdots & & & & & \frac{\partial F_p}{\partial y_p} \\ & & & I & & & & -I \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial \hat{y}_1} & \cdots & \frac{\partial F_{p+1}}{\partial \hat{y}_p} & & & & \end{array}\right). \tag{8.4}$$

This system is approximately twice the size of the "sequential" system, but after doing some algebraic operations on this system, it can be rewritten in a more compressed form as:

$$\tilde{J}_E \begin{pmatrix} \delta x \\ \delta \hat{y}_1 \\ \delta \hat{y}_2 \\ \vdots \\ \delta \hat{y}_p \end{pmatrix} = \begin{pmatrix} -\frac{\partial F_1}{\partial y_1}(\hat{y}_1 - y_1) \\ -\frac{\partial F_2}{\partial y_2}(\hat{y}_2 - y_2) \\ \vdots \\ -F \end{pmatrix} \tag{8.5}$$

where

$$\tilde{J}_E = \left(\begin{array}{c|cccc} \frac{\partial F_1}{\partial x} & \frac{\partial F_1}{\partial y_1} & & & \\ \frac{\partial F_2}{\partial x} & \frac{\partial F_2}{\partial \hat{y}_1} & \frac{\partial F_2}{\partial y_2} & & \\ \vdots & \vdots & & \ddots & \\ \frac{\partial F_p}{\partial x} & \frac{\partial F_p}{\partial \hat{y}_1} & \cdots & & \frac{\partial F_p}{\partial y_p} \\ \hline \frac{\partial F_{p+1}}{\partial x} & \frac{\partial F_{p+1}}{\partial \hat{y}_1} & \cdots & & \frac{\partial F_{p+1}}{\partial y_p} \end{array}\right). \tag{8.6}$$

## Comments

1. The functions $F_i$ and partials of $F_i$ can be computed in **parallel.**
2. The final system (8.5) looks of exactly the same complexity as the sequential structured system (8.1). There is a little extra complexity in forming the right hand side vector, see equation (8.5).

## Relation to multiple shooting methods

Clearly we have introduced parallelism at the expense of computing the correct function value itself by introducing "guesses" for the intermediate values. This method is very closely related to multiple shooting methods for solving boundary value problems in ODEs, e.g., consider the boundary value problem related to the ODE $dy/dt = f(y,t)$, and the goal is to compute $y(0)$ given $y(1) + y(2) = given$. The single shooting method makes a guess for the value of $y(0)$ and solves the initial value problem and then computes a new guess for $y(0)$ depending on the difference of $y(1) + y(2)$ from the given target. The multiple shooting method uses multiple guesses for $y$ at given time values $t_i, 1 \leq p, t_i \leq t_{i+1}$ and solves a bunch of IVPs from the guesses $y(t_i)$. It is well known that multiple shooting methods are generally more robust and well conditioned and as well as give rise to parallelism, since the multiple IVPs can now be solved in parallel. The idea of breaking up the general structured computation (which is sequential in nature) is exactly the same. In summary here is a list of potential gains that can be made by breaking up the structured computation into independent components:

- Parallelism.
- Well-conditioned Newton system.
- Reduction of nonlinearity – clearly the independent system includes all $F_i$ in an independent manner reducing the nonlinearity compared to the original $F$. For a trivial example consider the function $F(x) = x^4 - 1$. We can break it into two parts each of which compute only the second powers, e.g. $y = x^2, F = y^2 - 1$, in extended function notation $F_E(x,y) = 0$ shows up as

$$F_E = \begin{pmatrix} x^2 - y \\ y^2 - 1 \end{pmatrix}.$$

  $F_E(x,y)$ is just quadratic in its arguments, i.e. *less nonlinear* than $F$.

## Complexity

Let $C1$ be the cost of evaluating the function and the derivatives(the partials) and $C2$ be the cost of solving the Newton step. From the above observations, the systems we get for the sequential and the parallel case are similar(in terms of size and sparsity) and hence the cost for solving for the Newton step for the two cases will bethe same. However, the computation of function and extended Jacobian can be completely parallelized as we have illustrated before; hence we would expect close to full speedup in general, as long as we have number of processors less than the number of structured components, $nproc \leq p + 1$. Also for the full speedup we are assuming that all $F_i$ are of comparable costs.

In summary,

- Time required to compute the Newton step Sequentially $= C1 + C2$;
- Time required to compute the Newton step in parallel $= C1/nproc + C2$.

## 8.2 Parallelizing AD of Matrix-Vector Operations

We present some parallel implementation ideas here to parallelize the AD of MATLAB like the matrix-vector operations as presented in §4. We have developed these ideas keeping in mind the potential implementation in MultiMATLAB [TMC+96]. We consider the parallelization of both the forward and reverse mode, in particular, the tangent and adjoint products $(JV, J^TW)$ where $W \in \Re^{m\times p}$ and $V \in \Re^{n\times p}$.

We look at some basic matrix vector operations and comment on the parallelization aspect. The cost equations can be defined by using three variables: $N$ the size of the vector (or $[M, N]$ the size of the matrix); NUMPROC the number of processors; and $p$ the the column dimension of the product.

For the purpose of this section we make a basic assumption about parallel distribution of the value and the derivative data. A vector-valued variable has its value distributed equally among the processors, and its derivative $n \times p$ has each column distributed among the processors. We can also look at other more complex ways of distribution of data among processors, e.g., block distribution etc., however, that will be a digression from the main point here which is to see how parallelism can arise in forward and reverse mode which might not be present in the function evaluation itself.

Here are the basic subset of parallel matrix-vector operations, the cost equations are based on:

- **Add($n$):** Stands for the addition or subtraction operation between two $n$-vectors.
- **Dot($n$):** Dot product of two $n$-vectors.
- **Mul($n$):** Multiplication (element by element) of two $n$-vectors.
- **Scale($n$):** Multiplication of an $n$-vector by a scalar. For all purposes this operation is equivalent to Mul, so we will use Mul to represent this operation.

It is trivial to notice that Add, Mul and Scale operations are embarrassingly parallel, but the operation Dot requires some reduction operation among the processors. All matrix-vector operations can be broken down in terms of these basic operations. The parallelism shown by a particular matrix-vector operation can be observed by how many Dot operations it contains.

## Parallelization of basic forward and reverse modes

- $z = x^T y$

  Forward: $\dot{z} = y^T \dot{x} + x^T \dot{y}$.

  Reverse: $x_d^* + = y * z_d^*, \quad y_d^* + = x * z_d^*$

  Implementation of the forward mode is similar to implementation of the computation itself. The adjoint computation, however, is embarrassingly parallel. Consider the case where $x$ and $y$ are vectors, and $z$ is a scalar – the adjoint computation is just a scaling operation, which can be carried out trivially in parallel, compared to the dot product operation.

  Here are the costs of these operations:

  1. Function evaluation = 1 dot($n$)
  2. Forward Mode = $2 * p$ dot($n$) + $p$ add($n$)
  3. Reverse mode = $2 * p$ mul($n$)

  Clearly the reverse mode is cheaper in terms of number of floating point operations and is also embarrassingly parallel.

- $z = x + y$

  Forward: $\dot{z} = \dot{x} + \dot{y}$.

  Reverse: $x_d^* + = z_d^*, \quad y_d^* + = z_d^*$.

  Both reverse and forward operations are embarrassingly parallel.

  1. Function evaluation = 1 add($n$).
  2. Forward Mode = $p$ add($n$).
  3. Reverse mode = $2 * p$ add($n$).

- $z = x. * y$

  Forward: $\dot{z}(:, i) = \dot{x}(:, i). * y + \dot{y}(:, i). * x$.

  Reverse: $x_d^* + = diag(y) * z_d^*, \quad y_d^* + = diag(x) * z_d^*$.

  1. Function evaluation = 1 mul($n$).
  2. Forward Mode = $2 * p$ mul($n$)+ $p$ add($n$).
  3. Reverse mode = $2 * p$ Scale($n$).

  Again both the forward and reverse modes are embarrassingly parallel.

- $y = Ax$

  Forward: $\dot{y}(:,i) = \dot{A}(:,:,i)x + A\dot{x}(:,i)$.

  Reverse: $x_d^* + = A^T * y_d^*, \quad A_d(:,j,:)^* + = x(j) * y_d^*$.

  Depending on the shape of $A$, the parallelism in reverse and forward mode operations can be compared. E.g., if A is a single row, then this operation is equivalent to the dot product which we have discussed earlier. If A is square, both are equivalent to a square matrix times a vector. If A has a single column, i.e. $x$ is a scalar, then forward mode is embarrassingly parallel but reverse mode is a dot product. This is the opposite of the dot product operation where the reverse mode was embarrassingly parallel.

  More generally, consider $A \in \Re^{m\times n}, x \in \Re^n$ then the cost of forward and reverse mode operations can be written down as:

  1. function evaluation: $m$ dot($n$).
  2. forward mode: $2 * m * p$ dot($n$)+ $p$ add($n$) (Recall that all rows of $A$ are assumed distributed equally among processors).
  3. reverse mode: $2 * n * p$ dot($m$) + $n * p$ scaling operations (Suppose that all columns of $A$ are assumed distributed equally among processors for the multiply operation with $A^T$).

  Depending on the distribution of $A$ and the relative values of $m$ and $n$ the performance and parallelism in forward and reverse modes can be assessed.

- $C = A + B$

  Forward: $\dot{C} = \dot{A} + \dot{B}$.

  Reverse: $A_d^* + = C_d^*, \quad B_d^* + = C_d^*$.

  Both are embarrassingly parallel.

- $C = A * B$

  Forward: $\dot{C}(:,:,i) = \dot{A}(:,:,i) * B + A * \dot{B}(:,:,i)$.

  Reverse: $A_d(:,:,i)^* + = C_d^*(:,:,i) * B^T, \quad B_d(:,:,i)^* + = A^T * C_d^*(:,:,i)$.

  Again depending on the dimensions of $A$, the forward mode or reverse mode's parallelism can be different.

- $C = A. * B$

  Forward: $\dot{C}(:,:,i) = \dot{A}(:,:,i). * B + A. * \dot{B}(:,:,i)$.

  Reverse: $A_d(:,:,i)^* + = C_d(:,:,i)^*. * B, \quad B_d(:,:,i)^* + = C_d(:,:,i)^*. * A$.

  Both the forward and reverse mode operations are embarrassingly parallel, here is a breakdown of the costs involved.

  1. Function evaluation $= m$ mul($n$).

2. Forward Mode $= 2 * p * m$ mul($n$).
3. Reverse mode $= 2 * p * m$ mul($n$).

The computation of the Jacobian and Hessian sparsity pattern can also be parallelized similarly and will involve sparse parallel linear algebra.

# Chapter 9

# Conclusions

The work presented in this thesis fits in a layered organization, as shown in Figure 9.1.

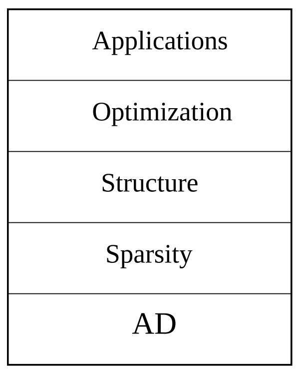



Figure 9.1: Layered view

In summary, we have made contributions to the top four layers in this framework, which can be collectively seen as adding a layer of "intelligence" to the bare AD tool. The ideas presented in this work often provide an order of magnitude savings in computational cost sometimes moving a solution process from infeasible to feasible.

This layer of "intelligence" consists of many different tools, e.g., exploiting sparsity is one "tool" in this layer which works independently of the bare AD tool (but makes the computation of derivatives using AD cheaper both in time and space); "structure exploitation" is another tool which works on top of both sparsity exploitation and the bare AD tool.

We have also proposed other tools in this layer, e.g., AD of matrix vector operations which work independently of both the sparsity and the structure exploitation tools. We present a detailed treatment of some applications and spell out the structure in detail to give insight into the efficient application of these techniques in conjunction with the AD tool. We also provide other plug-ins in this layer, e.g., scope for parallel implementation at the structure level or the matrix vector level and borrowing the concepts from this layer to enhance the performance of optimization algorithms themselves, e.g., preconditioning and the technology to handle the implicit computations. We have also provided the users of AD some insight into the intelligent use of AD technology.

We conclude with a brief mention of other aspects of AD technology that may be relevant to some applications. In particular, we show how to compute higher order derivatives (2nd order and higher) using AD. We also present some non-trivial caveats – instances when AD won't work or instances where AD needs to be applied carefully. Finally we mention some related work in AD technology which is relevant to the work presented in this thesis.

## 9.1 Computation of Higher Order Derivatives

For the purpose of this section, assume $f \in \Re^n \to \Re$ is a nonlinear scalar function. We consider the complexity of computing 2nd and 3rd order derivatives using AD.

### 9.1.1 Computing derivative-matrix products

**Computing** $HV$

For computing a Hessian matrix product, we can combine the forward and the reverse modes in the following fashion:

1. Compute $U(x) = \nabla f(x)^T V$ using the forward mode. Then $U \in \Re^p$. and $\omega(U) = p \cdot \omega(f), S(U) = p \cdot S(f)$.

2. Compute $HV = (\frac{dU(x)^T}{dx})^T$ by the reverse mode. Then $\omega(HV) = p^2 \cdot \omega(f)$, $S = numIvars(\nabla f(x)^T V) = p \cdot numIvars(f)$.

In fact we can swap the application of the reverse and forward modes:

1. Compute the gradient $W(x) = \nabla f(x)$ using the reverse mode. Then $W \in \Re^n$, $\omega(W) = 2\omega(f), S(W) = numIvars(f)$.

2. Compute $HV = (\frac{dW(x)}{dx}) * V$ by the forward mode. Then $\omega(HV) = 2 \cdot p \cdot \omega(f)), S = p \cdot numIvars(f)$.

This option of using the reverse mode first saves a factor of $p$ in the temporal complexity, but the spatial complexity is the same.

**Computing** $V^T HV$

It turns out we can avoid using the reverse mode with two applications of forward mode.

1. Compute $U(x) = \nabla f(x)^T V$ using the forward mode. Then $U \in \Re^p$, $\omega(U) = p \cdot \omega(f), S = p \cdot S(f)$

2. Compute $V^THV = \frac{dU(x)^T}{dx} * V$ by the forward mode again. $\omega(V^THV) = p^2 \cdot \omega(f), S = p^2 \cdot S(f)$

The space complexity of $V^THV$ is much better than that of $HV$ since the reverse mode is not used.

**Computing $\nabla^3 f(x) * v_1 * v_2$**

Assume $v_1$ and $v_2$ are vectors. This is easy to generalize it to the case where $v_1$ and $v_2$ are matrices. The most cost effective way of computing this tensor product is by the application of two forward modes and a single reverse mode. These applications can be done in any order, so there are a total of three possible recipes. We present here the one which applies the reverse mode in the last step.

1. Compute $U(x) = \nabla f(x)^T v_1$ using the forward mode; $U \in \Re$.
2. Compute $W(x) = v_1^T H v_2 = \frac{dU(x)}{dx} * v_2$ by the forward mode; $W \in \Re$.
3. Compute $\nabla^3 f(x) * v_1 * v_2 = \nabla(W(x))$ by the reverse mode.

For a more general discussion on computing higher order derivatives, refer to paper by Andreas Griewank on computation of higher order derivatives via propagation of univariate Taylor series [GU95].

In Appendix A we present the computation of sparse tensors (extension of functionality from sparse Jacobian and Hessian matrices) via AD using graph theoretic techniques. The recipes presented above are utilized for the tensor computation.

## 9.2 AD Caveats

*Does AD work on any computer code representing a differentiable mapping?* The answer is no. In this section we illustrate cases when AD might not work correctly.

### 9.2.1 AD of table look-up functions

A class of functions where AD won't work "as expected" are what we call the table-lookup functions. These functions are presented not as continuous transformations but as discrete table look-ups. To illustrate, consider the following function which computes $y = x^3$:

```
function y = cube(x)
if x == 1
    y = 1;
else
    y = x^3;
end
```

AD won't produce the right derivative $dy/dx$ for $x = 1$, since the dependence of $y$ on $x$ doesn't show up for $x = 1$ and it will return derivative $= 0$. On the other hand finite differences will return a good approximation.

Another example is given in the following code which computes $z = x * y$:

```
function z = times(x, y)
if x == 1
    z = y;
elseif y == 1
    z = x;
else
    z = x * y;
end
```

For $y = 1$, or $x = 1$, AD won't return the correct output. E.g., if $x = 1, y = 5, dz/dx$ will be returned as 0.

In general AD will fail on functions which involve any kind of table look-up as a part of function computation. E.g., the "cube" function presented before uses partial table look-up (at $x = 1$) and it can be shown that for partial table (table look-up only at discrete set of points) look-up functions FD works but AD doesn't. One could also think about pure table look-up functions where the function is represented by a set of pairs of values $(x_i, f(x_i), \quad i = 1 : p)$ as the one shown below for the function $f(x) = x^2$ at points $x_i = 1, 2, 3, \ldots$. For these functions neither AD nor FD will work.

```
function z = tablefun(x)
if 0 ≤ x < 1
    z = 0;
elseif 1 ≤ x < 2
    z = 1;
elseif 2 ≤ x < 3
    z = 4;
elseif 3 ≤ x < 4
    z = 9;
elseif ...
end
```

The derivatives returned will be zeros both for FD and AD. FD will experience glitches on the integer boundaries. In summary, AD works perfectly only for functions programmed as continuous transformations. In other words AD cannot "guess" the function as a continuous transformation if it is provided with only function values at certain points in the domain.

### 9.2.2 About derivatives w.r.t. continuous functions

In this section we illustrate a common mistake users of AD make when taking derivatives of an approximation of a continuous solution by discretizing the domains on which it is defined; typically the setting is the discretized solution of PDEs.

For example, consider a functional $f$ on continuous space $S = [0 \quad 1]$ and suppose we want to compute the derivative w.r.t. $s$ at a point $s \in S$. Typically $f(s)$ is computed by first computing $f(S)$ over the whole space $S$ (discretized appropriately) and then returning the value at point $s$. A template of the function you might employ to compute $f(s)$ is shown in Figure 9.2.

```
function z = f(s)
N = 100; S = linspace(0, 1, N);
[ Compute the discrete evaluation: Fval = f(S)
Fval will be a vector of length N ]
z = Fval(S(s * N));
```

Figure 9.2: A template to compute $f(s)$

This is a completely valid way to compute $f(s)$, but AD won't work on such a computation, since the discretized output $Fval$ doesn't depend on the *independent* variable $s$. This is because $Fval$ is a function of discretized space $S$ which is a constant (i.e., independent of $s$). Hence AD will always return the derivative equal to 0. However, finite difference will work; this type of function evaluation can be seen as table look-up function as $Fval$ can be seen as a table of constant values corresponding to different values of $s$.

The solution is to treat the whole space $S$ as the independent variable, which conceptually makes sense, since that is what we want; the derivatives w.r.t. the $S$ domain.

### 9.2.3 AD of the math Vs AD of the code

In this section we present a positive result about AD and illustrate some results about using AD for numerical integration schemes.

A lot of computations dealing with continuous physical models are actually carried out numerically on a grid via a finite difference or finite element method. Typically, these programs do not compute the mathematical function of interest, but rather a computational approximation to that function. A fundamental question is, then, how the derivatives of such approximate computer models (as they are computed via AD) relate to the derivatives of the mathematical function of interest. Specific issues include the following:

- What is the convergence behavior of derivatives of programs employing an iterative scheme? This topic has been studied extensively by Griewank and others [GBC$^+$93].
- What are suitable methods of differentiating programs with discontinuities? How do we define sensible derivatives for systems exhibiting shocks or discontinuities?

- The derivatives of a discretized function can be more or less "smooth" than the derivatives of the original function. One consequence is that discretizing, then differentiating may not produce the same results as differentiating, then discretizing. Is it always better to apply AD to a discretized function than to discretize the derivatives? We provide some insight into this specific topic in this section.

Bischof and Eberhard [EB96] present some basic motivation into this problem, by discussing the numerical solution of an ODE. The major issue is the making the choice of "differentiate first" or "discretize first" for computation of the derivative as mentioned in the third issue above . These two choices are illustrated in Figure 9.3.

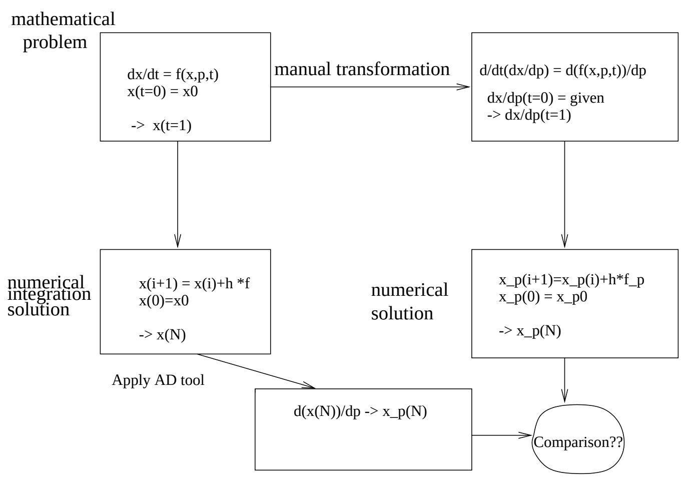



Figure 9.3: Automatic differentiation versus manual transformation

These two are really different choices specially if you take choice of first differentiating the ODE and then integrate using a different mesh/stepsize to compute the derivative. As long as the two meshes (or in the case of 1-D ODEs, the stepsizes) are the same and do not depend on the independent parameter, we can show that these two choices are really equivalent. Consider a simple ODE for the analysis as shown in equation (9.1):

$$\frac{dy}{dt} = F(y, p, t). \tag{9.1}$$

Let us denote the differentiated ODE for the choice 2 by ODE' which can be written as in equation (9.2).

$$\frac{dy'}{dt} = F_p(y, p, t) + F_y(y, p, t)y' \tag{9.2}$$

In fact, the ODE for the derivative computation should combine the function and derivative evaluation and we redefine ODE' as shown in equation (9.3).

$$\frac{d\begin{pmatrix} y \\ y' \end{pmatrix}}{dt} = \begin{pmatrix} F(y,p,t) \\ F_p(y,p,t) + F_y(y,p,t)y' \end{pmatrix}. \tag{9.3}$$

We want to show that AD of the numerical solution of ODE is equivalent to a numerical solution of ODE' provided that we use the same integration method and the same constant step size. In particular, we provide insights into and answers of the questions related to order and stability of the method for solution of derivatives. For basic information about finite difference integration methods for ODEs and PDEs and their order and stability refer to Trefethen [Tre97]. Note that this section deals with computation of derivatives in the forward mode. For the computation of adjoints similar adjoint differential equations can be derived, for example refer to [Ebe96].

- **Equivalence of methods**

  We first show that all LMS integration formulas and all linear formulas for integration of PDEs (like the upwind or the Lax-Frederics method) for ODE', are equivalent to applying AD of same LMS procedure for ODE, as long as the step sizes $(dh, dt)$ are kept constant. This is easy to show. Consider a general LMS integration method for ODE (equation (9.1)):

  $$\sum_{j=0}^{s} \alpha_j y^{n+j} = h \sum_{j=0}^{s} \alpha_j F(y^{n+j}, h(n+j), p) \tag{9.4}$$

  Let's now write down the LMS method for ODE' as shown in equation (9.5) just showing the propagation of the derivatives.

  $$\sum_{j=0}^{s} \alpha_j y'^{n+j} = h \sum_{j=0}^{s} \alpha_j (F_p(y^{n+j}, p, h(n+j)) + F_y(y^{n+j}, p, h(n+j))y'^{n+j}). \tag{9.5}$$

  Now let's consider differentiation of equation (9.4) w.r.t. $p$ which will correspond to AD of the numerical method to integrate ODE, as shown in equation (9.6).

  $$\sum_{j=0}^{s} \alpha_j dy^{n+j} = h \sum_{j=0}^{s} \alpha_j F_p(y^{n+j}, h(n+j), p)dp + \alpha_j F_y(y^{n+j}, h(n+j), p)dy^{n+j}. \tag{9.6}$$

  It is trivial to see that the equations (9.5) and (9.6) are equivalent.

- **Stiffness**

Consider the Jacobian of ODE' w.r.t. $y, y'$, i.e. differentiate the nonlinear equations:

$$\begin{pmatrix} F(y,p,t) \\ F_p(y,p,t) + F_y(y,p,t)y' \end{pmatrix}. \tag{9.7}$$

We get equation:

$$J' = \begin{pmatrix} F_y(y,p,t) & 0 \\ F_{py}(y,p,t) + F_{yy}(y,p,t)y' * y' & F_y(y,p,t) \end{pmatrix} \equiv \begin{pmatrix} J & 0 \\ M & J \end{pmatrix}. \tag{9.8}$$

Since $J'$ is a block triangular matrix and $J$ appears on the diagonals, $J'$ will have the same set of eigenvalues as $J$ and hence the same stiffness properties. So in a way ODE' and ODE are similar systems.

- **Order and Stability:** It is natural to ask how the order and stability of the solution procedure for ODE and ODE' compare.
  - **Order**
    The order of accuracy of the LMS method used for ODE will be the same on ODE' since the order depends just on the method used. Hence if LMS method has order $q$ for ODE it will have the same order $q$ for ODE'.
  - **Stability of LMS** The characteristic polynomials of the LMS formulas are defined by

    $$\rho(z) = \sum_{j=0}^{s} \alpha_j z^j \quad \sigma(z) = \sum_{j=0}^{s} \beta_j z^j \tag{9.9}$$

    The stability of the linear multistep formulas depends solely on the roots of polynomial $\rho(z)$ and not on the underlying differential equation involved. Hence if a LMS formula is stable for ODE it will be stable for ODE'.
  - **Stability Regions** This is connected with A-stability of the ODEs and used to defined the stable regions for the step size $h$. For LMS formulas the stability regions are defined by the concept of absolute stability.

    **Result 9.2.1** *A LMS formula is absolutely stable for a particular value of step size $\bar{h}$ if and only if all zeros of $\rho(z) - \bar{h}\sigma(z)$ satisfy $|z| \leq 1$ and any zero with $|z| = 1$ is simple.*

    Hence the stability regions of ODE and ODE' will be the same.
- **Dahlquisht stability for PDEs** Similarly we can consider PDE and PDE' along with a set of general finite difference schemes for solving PDEs like upwind etc. It can again be shown on the same lines that for the linear finite

difference schemes like upwind, Lax-Fredericks, Crank-Nicholson etc, Differentiation of PDE is the same as the same finite difference method applying to PDE'. We ask the question about Dahlquist stability of particular methods of integrations w.r.t. PDE and PDE'.

For illustration we consider the nonlinear hyperbolic PDE shown in equation (9.10):

$$u_t = F(u,p)_x. \tag{9.10}$$

$$\begin{pmatrix} u_t \\ (u_p)_t \end{pmatrix} = \begin{pmatrix} F(u,p)_x \\ F_p(u,p)_x + (F_u(u,p))_x u_p \end{pmatrix}. \tag{9.11}$$

PDE' is shown in equation (9.11)

The amplification factor for PDE using the frequency domain method is given by $g(\chi)$. The Von Neumann condition says that the finite difference method is stable only if

$$|g(\chi)| < 1 + O(h). \tag{9.12}$$

Now the amplification factor of PDE' is given by $g'(\chi)$ which is a matrix of the form

$$g'(\chi) = \begin{pmatrix} g(\chi) & 0 \\ M(\chi) & g(\chi) \end{pmatrix}$$

where $g(\chi)$ is the amplification factor for PDE. For the vector formulas with amplification matrix $G$ the Von Neumann condition says

$$\rho(G(\chi)) < 1 + O(h) \tag{9.13}$$

where $\rho(.)$ the spectral radius or largest of the moduli of eigenvalues of matrix $G(\chi)$. Now since $g'$ is a block triangular matrix with $g$ on the diagonal, if $g(\chi)$ satisfies the condition (9.12) then $g'(\chi)$ can be shown to satisfy (9.13).

We can also prove similar results about order of accuracy and stability regions with PDEs.

In summary if a integration method M is good for ODE(PDE) with a parameter $p$, then it is also good for ODE'(PDE'). Intuitively it is expected to work fine since ODE' is linear in the derivative (hence less nonlinear than ODE).

## 9.3 Related Work

Recently, there has been a lot of work in AD and especially in applications which use AD. Here we present references ranging from elementary work on automatic differentiation theory to the AD tools that are in existence to the work on the use of AD for optimization. We have subdivided this section into relevant categories to organize the references properly.

## Automatic differentiation theory

Automatic differentiation as a technique has been known for over 20 years, but most of the action has been taking place in last five to seven years. Andreas Griewank has been a pioneer in the recent development of AD. For a historical survey of automatic differentiation and related areas one may consult the article by Masao Iri [Iri91], in the proceedings of the Breckenridge conference in 1991 [GC91]. For a survey on AD tools, refer to [Jue91].

## Sparsity/Structure exploitation using AD

Sparse Jacobians using only the forward mode of AD were discussed in[AMB+94]. Structured gradients using the forward mode and exploiting partial separability were addressed in [BBKM95]. There has also been work in exploiting general structure for computation of gradients using AD [CJ97]. Griewank and others have looked at Newsam and Ramsdell approach to compute the sparse Jacobian matrix [NR83, GUG96,GU96].

There has been work on using both modes of automatic differentiation to exploit sparsity in Jacobian matrices [CV98c,SH95].

## Automatic differentiation tools

The proceedings of the Breckenridge conference in 1991 [GC91] contains an article by David Juedes of then existing software packages [Jue91]. There has been phenomenal development in AD tools since then involving both improvement of then existing tools and new AD tools. Some of the prominent AD tools now are ADIFOR [BCC+92], ADOL-C [GJU96] and ODYSSEE [RDG93]. In our work, we are developing ADMAT, an AD tool for M-files[CVa].

## Optimization/computing environments using AD tools

The NEOS server [CMM97] is a prime example of an optimization WWW server employing AD. ADMIT is an ongoing project at Cornell University and is a part of the work presented in this thesis. LSOT [Col] has also been tried out in conjunction with ADMIT.

## Solution of real-world problems using AD

Personnel at Argonne National Labs have been involved in applying AD (in particular, ADIFOR) on a variety of problems in computational science, including Navier-Stokes computations [HMB97], weather models [BPK95], groundwater transport models [BWS+94], aircraft design [BGHK94] and other applications. Coleman, Santosa and Verma have looked at a variety of inverse problems including medical imaging [CSV] and geophysical seismology problems [CSV97].

# Appendix A

# Computing Sparse Tensors

In chapter 2 we discussed the computation of sparse Jacobian and Hessian matrices. Three dimensional derivative objects also arise in a variety of computations; here we look at three different scenarios.

## A.1 Jacobian of a matrix function $A(x)$

Consider the matrix function $A(x)$ mapping $\Re^n \rightarrow \Re^{p \times q}$. The Jacobian matrix, represented $A'(x)$, is a tensor of size $p \times q \times n$. All the entries in this derivative tensor are independent, i.e., there is no symmetry to be exploited.

Assume that access to the matrix function $A(x)$ is available through matrix vector products of form $A(x)v$, i.e., there is an oracle which gives you $A(x)v$ when supplied the vector (or matrix) $v$. Using graph theory and combinatorial techniques similar to Coleman and Moré [CM84b], the tensor $A(x)$ can be recovered from tensor products of the form $A'(x)d1_i d2_i, i = 1 : r$, where we seek to minimize the number of products needed, i.e., $r$. The tensor product can be formed via AD using the recipes given previously in Chapter 9.

## A.2 Second derivative of a vector function $F(x)$ – partial symmetry

Consider the function $F(x)$ mapping $\Re^n \rightarrow \Re^m$. The second derivative $F''(x)$ is a tensor of size $m \times n \times n$. However we are going to orient it in a different way, i.e. as a $n \times n \times m$ tensor, and call it the modified version $F''_m(x)$, so that in the limiting case $m = 1$, it reduces to a Hessian. All the entries in this derivative tensor aren't independent, i.e., there is a symmetry, in particular $F''_m(x)(:, :, i) \forall i$ is a symmetric Hessian matrix of size $n \times n$. The tensor $F''_m(x)$ can be recovered from the tensor products of the form $F''_m(x)d1_i d2_i, i = 1 : r$. This solution technique also involves using graph coloring of adjacency graphs to exploit the partial symmetry, similar to Coleman and Moré [CM84a,CC86].

## A.3 Third derivative of a scalar function $f(x)$ – full symmetry

Consider the function $f(x) : \Re^n \rightarrow \Re$. The third derivative matrix $\nabla^3 f(x)$ is a tensor of size $n \times n \times n$ and has full symmetry, i.e. $\nabla^3 f(i,j,k) = \nabla^3 f(j,i,k) = \nabla^3 f(i,k,j) = \nabla^3 f(k,i,j) = \nabla^3 f(j,k,i) = \nabla^3 f(k,j,i)$. In particular, only about one-sixth of the nonzeros are independent. We describe here the technique to compute $\nabla^3 f(x)$ via tensor products of form $\nabla^3 f(x) * d1_i * d2_i, i = 1 : r$ using graph theoretic techniques which exploit the full symmetry.

We define an adjacency graph as follows. The tensor $\nabla^3 f$ has $n^2$ columns. The columns can be represented by $C_{j,k} = \nabla^3 f(x)(:, j, k), j = 1 : n, k = 1 : n$.

We define the adjacency graph with a vertex for every column: $\mathcal{G}(A) = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{(j,k), j = 1 : n, k = 1 : n\}$. There is an edge, $((a,b),(c,d)) \in \mathcal{E}$ if $b = d$ and there is a nonzero in positions $(a,c,d) = (c,a,d) = (a,c,b) = (c,a,b)$ (these positions are equivalent due to symmetry), similarly if $a = c$ and there is a nonzero in positions $(a,b,d) = (c,b,d) = (a,d,b) = (c,d,b)$.

This graph is defined like an adjacency graph, to take care of the full symmetry which shows up in the tensor. A nonzero in the tensor, $\nabla^3 f(i,j,k)$ (assume $i \neq j \neq k$), belongs to six different nodes: $(i,k)$, $(j,k)$, $(j,i)$, $(k,i)$, $(k,j)$, $(i,j)$. To be able to exploit the symmetry fully, we can make sure that we can extract nonzero $(i,j,k)$ from at least one of these six columns. For a nonzero of the form $(i,i,j)$, there are 3 vertices involved $(i,i),(i,j),(j,i)$, and for a nonzero of form $(i,i,i)$, there is only one column involved: $(i,i)$.

**Coloring and marking assignment.** Now we define a coloring and a marking assignment of the columns, using a generalization of the path coloring technique [CM84a]. The marking assignment helps to determine from which column a particular nonzero should be extracted. Marked columns are labelled with a set of row positions which denotes the set of nonzeros to be determined from this column. The important condition is that the columns having the same color don't intersect in the labelled row positions, since that won't allow us to determine the nonzeros uniquely.

1. Adjacent vertices have different assignments.

2. If $(a,b)$ and $(c,d)$ have a common color, then so do vertices $(a,d)$ and $(b,c)$.

3. For each edge, one and only one column corresponding to the edge is *marked* with a label being the row position of the nonzero in that column. For the nonzero $(i,j,k)$ the column marked is $(j,k)$ with the label $i$.

4. For the set of vertices having the same color, the marked vertices have no intersection with other columns in the set in the labelled row positions.

Based on this coloring we can define the direction pairs $(v_1, v_2)$ for each color, by collecting all the vertices $(a_1,b_1),\ldots,(a_s,b_s)$ assigned that color and define $v_1 = \sum_{i \in a_1,\ldots,a_s} e_i$ and $v_2 = \sum_{i \in b_1,\ldots,b_s} e_i$.

It can be shown that all the nonzeros can be recovered from this strategy. For any nonzero $(i, j, k)$ you want to recover, pick the column marked with the edge $(i, j, k)$, and look for the direction vector corresponding to the color the marked column in colored. Since there is no intersection in the position, the desired nonzero is located, we can uniquely extract the nonzero.

# Appendix B

# A conjecture for sparse matrices

## Introduction

Here we present a conjecture for lower bounds on the number of groups required for estimation of sparse Jacobian matrix and Hessian matrices. Interestingly, the formulation for lower bounds for Jacobians and Hessian matrices as presented here is very similar. Also, note that these lower bounds are for using a general elimination procedure on the lines of the Newsam & Ramsdell procedure [NR83] to compute sparse matrices $A$ via products of form $Av$.

## Some Notation

### Jacobians

**Problem:** We want to estimate $J$ from matrix vector products $JV$ and $W^T J$, $V \in \Re^{n\times p_2}$ and $W \in \Re^{m\times p_2}$. The objective is to minimize $p_1 + p_2$.

#### The r/c assignment and directed bipartite graphs

Consider assigning a binary token which is either "r" or "c" with every nonzero entry of the $J$. This operation of assigning "r" or "c" token is equivalent to associating directions with edges of $G_b(J)$, or *directing* the bipartite graph $G_b(A)$. If a nonzero $J_{ij}$ (or an edge $(i, j)$) is assigned "c", the edge $(i, j)$ is converted to $i \to j$, and if it is assigned "r", the edge $(i, j)$ is converted to $j \to i$.

The Figure B.1 shows the association for a sample $2 \times 2$ Jacobian matrix, with $(2, 1)$ entry zero:

**Definition B.0.1** *$\mathcal{A}(J)$ refers to a given r/c assignment to nonzeros in $J$.*

**Definition B.0.2** *$g(\mathcal{A}(J)) = \max_i(c_i) + max_i(r_j)$ where $c_i$ is the number of "c"s in row $i$ and $r_i$ is the number of "r"s in column $j$, according to the assignment $\mathcal{A}(J)$.*

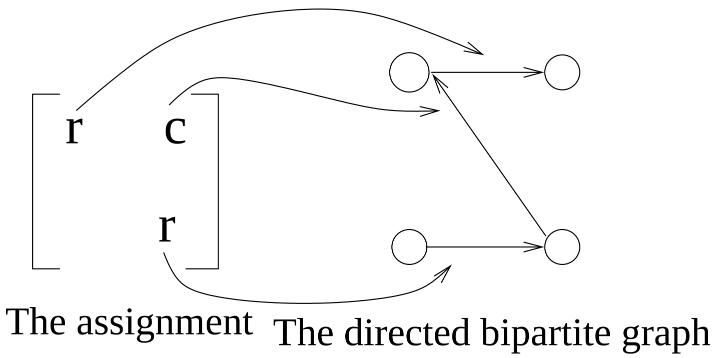



Figure B.1: From the matrix to the directed bi-partite graph

**Definition B.0.3** $g(J) = \min_{\mathcal{A}(J)}(g(\mathcal{A}(J))$, *i.e minimum over all possible r/c assignments.*

**Definition B.0.4** $g_{cyc}(J) = \min_{\mathcal{A}(J)}(g(\mathcal{A}(J))$, *i.e minimum over all possible r/c assignments* $\mathcal{A}$ *such that the corresponding directed graph is* acyclic.

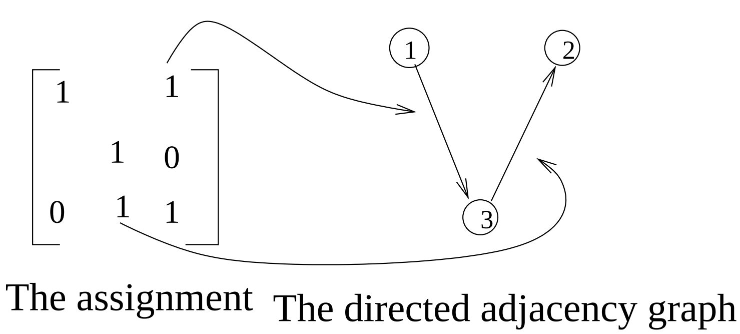
## Hessians

**Problem:** We want to estimate $H$ from $HV$, $V \in \Re^{n \times p}$ and minimize the value of $p$ which allows us to reconstruct the matrix $H$.

The Hessians are assumed to have nonzero diagonals.

### 0/1 assignments and directed adjacency graphs

Consider following definition of a 0/1 assignment for the nonzeros of the Hessian. Every element $(i, j)$, $\quad i \neq j$ is to be assigned either a 0 or a 1. Due to the symmetry, the $(i, j)$ element is same as the $(j, i)$ element and former element is assigned 1 if the latter is assigned 0 and vice-versa. This procedure is actually equivalent to *directing* the Adjacency graph, i.e. associating a direction with each edge in the adjacency graph of $H$, where there is a directed edge from $i$ to $j$ if $(i, j)$ is assigned 1, and from $(j, i)$ otherwise.




**Definition B.0.5** $\mathcal{A}(H)$ *refers to a given 0/1 assignment to the nonzeros in* $H$.

**Definition B.0.6** *$g(\mathcal{A}(H)) = 1 + \max_i(\rho_i)$ where $\rho_i$ refers to the number of 1's in row $i$ of the assignment $\mathcal{A}(H)$.*

**Definition B.0.7** *$g(H) = \min_{\mathcal{A}(H)} g(\mathcal{A}(H))$ i.e. the minimum is taken over all possible assignments.*

**Definition B.0.8** *$g_{cyc}(H) = \min_{\mathcal{A}(H)} g(\mathcal{A}(H))$ i.e. the minimum is taken over all possible assignments such that the corresponding directed graph is* acyclic.

## Derivation of a tight lower bound For Jacobians

Given matrices, $V \in \Re^{n \times p_1}, W \in \Re^{m \times p_2}, JV \in \Re^{m \times p_1}, J^T W \in \Re^{n \times p_2}$, it is possible to construct a system of size $(m * p_1 + n * p_2) \times nnz(J)$ which is to be solved for the nonzero entries of the Jacobian matrix.

The entries in A will come from the entries in $V$ and $W$, while those in the right hand side $r$ will have entries from $JV$ and $W^T J$ in it. Also, if we arrange the nonzeros of $J$ in a row-wise fashion, i.e. all the non zeros in row 1 first, and then second row unknowns etc, into the vector $nJ$(which is the unknown), the linear system we will get from the above ordering will look like

$$A \cdot nJ = r \tag{B.1}$$

**Properties of the system**

- The equations in the system can be divided into two parts, i.e

  $$A = [A_c; \quad A_r]$$

  where $Ac$ corresponds to forward product equations and $Ar$ corresponds to reverse product equations. And the way we have arranged the nonzeros, $Ac$ is in block matrix form, but $Ar$ looks more wide spread, actually it consists of column strips of size $p_2$.

- If we eliminate the nonzeros corresponding to first row, the rest of the system $\bar{A}$ is a subset of $A$, and corresponds to the structure of $\bar{J}$, which is $J$ with the first row removed. This is depicted in figure B.2.

**Definition B.0.9** *$LB$ = minimum number of groups $(p_1 + p_2)$ needed to solve for entries in the Jacobian matrix.*

**Result B.0.1** $g(J) \leq LB$

*Proof:* Skipped for brevity.

But is $g(J)$ a tight lower bound? Well, if we can show that the system of equation corresponding to assignment $g(J)$ is always nonsingular, then we are done

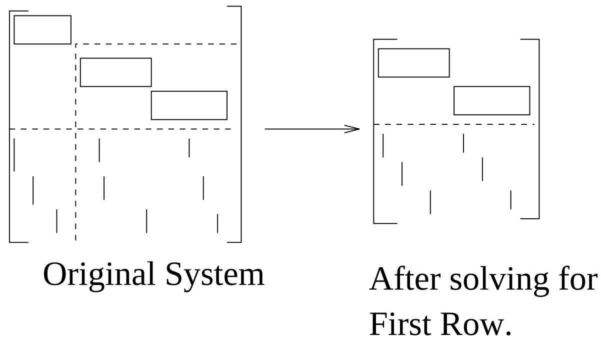



Figure B.2: Solving for the nonzeros

(i.e. $g(J)$ is indeed the tight lower bound). But this is not true, due to the following observation:

**Observation 1**: Submatrices where each row has $p_1$ "r"s and each column has $p_2$ "c"s cause a problem. The simplest example is (with $p_1 = 1, p_2 = 1$):

$$J = \begin{pmatrix} r & c \\ c & r \end{pmatrix}$$

It can be easily shown that system of equations corresponding to this is singular.

This observation is captured in the following theorem:

**Theorem B.0.1** *System is rank deficient iff we have a submatrix with all the rows having the same number of "c"s (say $p_1$) and all the columns having the same number of "r"s, in other words the graph has a subgraph which the same out degrees for rows and the same outdegrees for columns.*

*Proof:* Skipped for brevity.

Finally we present our conjecture for the tight lower bound:

**Conjecture:** $g_{cyc}(J)$ is a tight lower bound.

# Derivation of Bound for Hessians

We assume that $H$ has nonzero diagonal, this is the case in most problems. Since only about half the nonzeros are unknowns in $H$ (symmetry), we can take the ordering of variables in terms of the **upper triangular part of** $H$, i.e. all unknowns contained in upper triangular part. The ordering is row wise from row 1 down to row $m$.

A typical system is shown here as

$$A \cdot nH = r \tag{B.2}$$

where $A$ is the matrix constructed from the entries in $V$, $nH$ is the vector of nonzeros of $H$ which is laid out in the manner discussed above , and $r$ contains the corresponding entries from $HV$.

**Result B.0.2** *$g(H) \leq LB$.*

*Proof:* skipped for brevity.

**Observation:** A subgraph where each vertex has same outdeg causes problems. This example requires three groups to be solved, but the 0/1 assignment has $g(H) = 2$.

$$H = \begin{pmatrix} \times & \times & & \times \\ \times & \times & \times & \\ & \times & \times & \times \\ \times & & \times & \times \end{pmatrix} \qquad 0/1 assignment = \begin{pmatrix} 1 & 1 & & 0 \\ 0 & 1 & 1 & \\ & 0 & 1 & 1 \\ 1 & & 0 & 1 \end{pmatrix}$$

The system corresponding to 2 groups for above system is:

$$\begin{pmatrix} V_{11} & V_{21} & V_{41} & & & & & \\ V_{12} & V_{22} & V_{42} & & & & & \\ & V_{11} & & V_{21} & V_{31} & & & \\ & V_{12} & & V_{22} & V_{32} & & & \\ & & & & V_{21} & V_{31} & V_{41} & \\ & & & & V_{22} & V_{32} & V_{42} & \\ & & V_{11} & & & & V_{31} & V_{41} \\ & & V_{12} & & & & V_{32} & V_{42} \end{pmatrix} \begin{pmatrix} H_{11} \\ H_{12} \\ H_{14} \\ H_{22} \\ H_{23} \\ H_{33} \\ H_{34} \\ H_{44} \end{pmatrix} = \begin{pmatrix} (HV)_{11} \\ (HV)_{12} \\ (HV)_{21} \\ (HV)_{22} \\ (HV)_{31} \\ (HV)_{32} \\ (HV)_{41} \\ (HV)_{42} \end{pmatrix}$$

It can be shown that the system is singular!!! This observation is captured in following theorem .

**Theorem B.0.2** *The system becomes rank deficient iff we have a a subgraph of $G_D$ which has the same outdegree, say p, for all the vertices.*

*Proof:* Skipped for brevity.

Finally we present our conjecture for the tight lower bound:

**Conjecture:** Solution to tight lower bound problem is given by $g_{cyc}(H) = 1 + \max(deg(v))$

# What do these bound reflect?

- **Jacobians**.

  Since the graph $G_D$ is acyclic, we have some partial order on the nodes of the graph. This order is an order among the rows and columns of the matrix. This is exactly the process of partitioning as described in the bi-coloring partition of $J = (J_r, J_c)$. The directed acyclic graph corresponds exactly to the zigzag partition shown in the Figure B.3.

  Finally, we got a simple characterization of lower bound:

  **Result B.0.3** *$LB = \min(\rho(J_c) + \rho_(J_r))$ over all zigzag partitions $J = [J_c|J_r]$.*

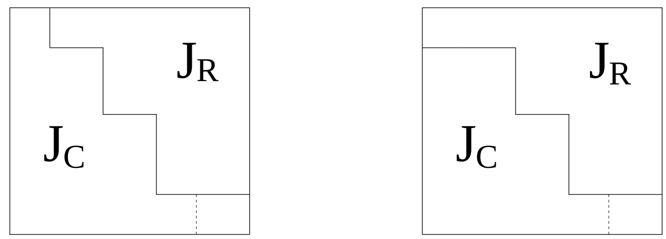



Figure B.3: Possible partitions of the matrix $\tilde{J} = P \cdot J \cdot Q$

The proof follows a corollary to Theorem B.0.1.

- **Hessians**.

  Since the graph $G_D$ is acyclic there exists a partial order on the vertices, in other words there is a permutation of $r_1, r_2, \cdots, r_m$ call it $P$, and the outdegree of vertex $i$ is exactly the number of nonzeros in the lower triangular part of $PHP^T$. So $LB = \rho(L)$ where $L =$ the lower triangular part of $(PHP)$.

  **Result B.0.4** *$LB = \rho(lower(PHP^T))$ over all permutations $P$, and lower extracts the lower triangular part of the Matrix.*

The proof follows a corollary to Theorem B.0.1.

# Appendix C

# Software

## C.1 ADOL-C

ADOL-C is a C++ package which facilitates the evaluation of first and higher order derivatives of vector functions that are defined by computer programs written in C or C++.

ADOL-C uses the object oriented features of C++, in particular it uses overloading of elementary functions to propagate derivative information along with the C++ function evaluation.

You can find details about the ADOL-C software(and download the latest version) from http://www.math.tu-drsden.de/˜ adol-c

## C.2 ADMIT-1

### Introduction

ADMIT-1 enables you to compute *sparse* Jacobian and Hessian matrices, using automatic differentiation technology, from a MATLAB environment. You need only supply a M-function to be differentiated and ADMIT-1 will exploit sparsity if present to yield sparse derivative matrices (in sparse MATLAB form). ADMIT-1 also allows for the calculation of gradients and has several other related functions.

For complete information on ADMIT-1, please refer to the ADMIT-1 user guide [CV97] and the software paper [CV98a].

### An example

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

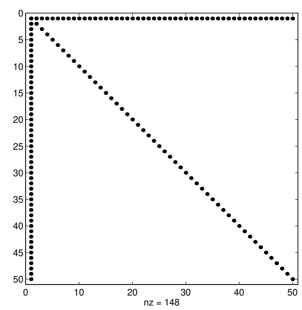



Figure C.1: The sparsity structure of Jacobian $J$

**function** $f\ =\ \mathit{examplefun}(x, m, \mathit{Extra})$
$f = x. * x;$
$f(1) = f(1) + x' * x;$
$f = f + x(1) * x(1);$

Assume this program is saved in file myfun.m. 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;
>> [f,J]=evalJ('myfun',x);
>> f
f =
     7
     2
     2
     2
     2
>> spy(J) <- Sparsity structure is displayed.
```

## Software design of ADMIT-1

The design of the ADMIT-1 toolbox is as shown in Figure C.2. 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 sparse techniques for the computation of sparse Jacobian and Hessians (and other derivative information like gradient and Jacobian matrix products etc.).

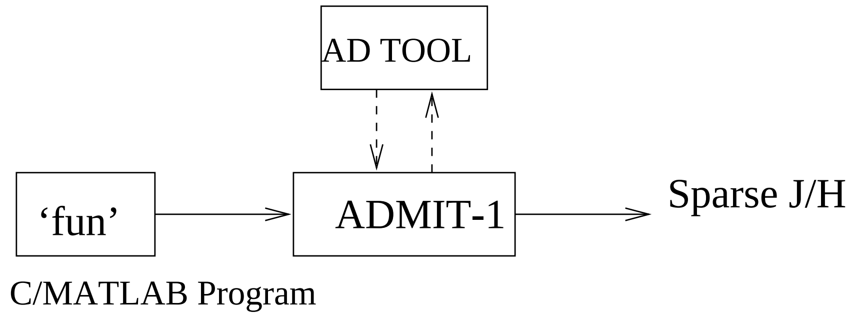



Figure C.2: Design of ADMIT-1 toolbox

### The ADMIT functions

## 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,J]=evalJ(fun,x,...)

## 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. [CC86,CM84a]).

### Synopsis

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

### Design of the user function: "fun"

Design your target M-function as follows.

```
function f= examplefun(x,m,Extra)

%crunch
```

```
end
```

The input argument `x` is a vector of dimension `n`; `f` is the output vector of dimension `m`. `Extra` is an additional parameter for use in the function.

# C.3 ADMIT-2

ADMIT-2 provides efficient methods to work with Jacobians and Hessians of "structured functions" from a MATLAB environment, employing the underlying theme of EASE (**E**xtended functions **A**nd **S**tructure **E**xploitation). It is built on top of ADMIT-1. You only need to supply the "structured function" to be differentiated and ADMIT-2 can **efficiently** compute derivative information in form of the extended Jacobian/Hessian matrices.

For complete information on ADMIT-2, please refer to the user guide [CV98b](in preparation).

## Classes of structured functions in ADMIT-2

ADMIT-2 can work with structured computations belonging to the following five different classes:

- General structured functions (class GF).
- Generalised partially separable (GP).
- Generalised composite functions (GC).
- Inverse problems (GI).
- Discrete-time optimal control problems (DO)

## Getting Started

### Example: A simple BVP

This problem falls into our "Inverse problem" class, GI. The problem is to solve a BVP in a parameter-less ODE shown in equation (C.1):

$$\frac{dy}{dt} = F(t,y) = y. * y + ty \tag{C.1}$$

**Problem:** Solve $y(0)$ given $y(1) = y_{given}$. For this simple example, we use a constant step size Euler's integration method.

The timestepping (Euler's) integration method (assumed to be in file `timestep.m`) in ADMIT2 notation is shown in Figure C.3.

$$
\begin{array}{|l|}
\hline
\textbf{function } y = \text{timestep}(x, m, Extra) \\
\%\ \textsf{Recover Information about step size, current time etc.} \\
\quad h = Extra(4); t = Extra(5); \\
\%\ \textsf{Compute the next state.} \\
\quad y = x + h.*(t.*x + x.*x); \\
\textbf{end} \\
\hline
\end{array}
$$

Figure C.3: The Euler's timestepping scheme

Here we show how to apply ADMIT-2 for computing the derivative information for the abovementioned boundary value problem:

- **Computing the extended Jacobian**

```
>> x=ones(10,1);
>> y_given=2*ones(10,1);
>> fdata = GIFdata('timestep',n,[],[],y_given);
>> [f,extJ]=evalJExt('GI','timestep',x,fdata) ;
```

- **Computing the Newton step and true Jacobian**

```
>>  ..
>>  ..
>> [f,extJ,ns,J]=evalJExt('DI','timestep',x,fdata) ;
```

## Software design

ADMIT-2 is designed to be easy to use; the input programs are M-files. ADMIT-2 can provide derivative information for large-scale structured problems without requiring the explicit formulation of derivative matrices. The ADMIT-2 design is shown in Figure C.4.

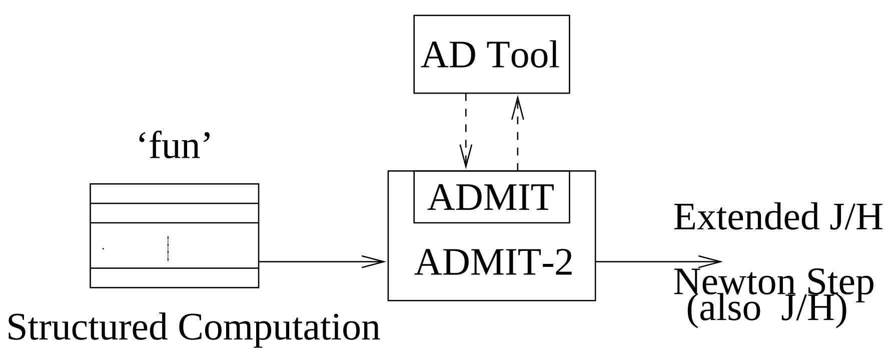



Figure C.4: ADMIT-2 design

## Functionality of ADMIT-2

The following ADMIT-2 "methods" can work with structured functions to return the required derivative information.

- **evalJExt:** Computes the extended Jacobian, the Newton step and the actual Jacobian (if wanted).
- **evalHExt:** Computes the extended Hessian, the Newton step and the actual Hessian (if wanted).
- **ExtJV:** Compute $JV$.
- **ExtWJ:** Compute $W^T J$.
- **ExtHV:** Compute $HV$.
- **ExtVHV:** Compute $V^T HV$.
- **ExtJfunc:** Computes a variety of information (like LU, QR factorization) relating to $J$ from the extended Jacobian matrix.
- **ExtHfunc :** Computes a variety of information (like the modified newton step, Cholesky factorization) relating to $H$ from the extended Hessian matrix.

## Main ADMIT2 functions

## evalJExt

### Purpose:

For computing the function, extended Jacobian matrix, Newton step and Jacobian matrix of a structured vector mapping.

### Synopsis

```
[f,ExtJ,ns,J]=evalJExt(class,fun,x,...)
```

## evalHExt

### Purpose:

For computing the function, gradient, extended Hessian matrix, Newton step and Hessian matrix of a structured vector mapping.

### Synopsis

```
[f,ExtH,ns,H]=evalHExt(class,fun,x,...)
```

## C.4 MATLAB AD tool

The motivation is to be able to build an AD tool for a high-level language like MATLAB. Thinking about AD in terms of high-level matrix-vector operations helps, especially in terms of storage requirements in the reverse mode, where you have to just save the high level vectors. There are other insights gained by this high-level view, e.g. information about parallelization of derivative code.

This is the first ever AD tool written for differentiating M-files. This tool belongs to the "operator overloading" class of AD tools and uses MATLAB5's OOP (Object Oriented Programming) feature for implementation.

For complete information on the ADMAT AD tool please refer to the user guide [CVa].

# Appendix D

# Using "Overloading" to compute objects other than derivatives

We have seen how to compute derivatives (e.g. gradients, Jacobians, Hessians) and in general arbitrary order derivatives using propagation of intermediate derivatives computationally using the chain rule. We have also seen how to compute sparsity patterns of Jacobians and Hessians by propagation of intermediate sparsity patterns. In this appendix, we present other computational objects (or properties) related to a function $y = F(x)$, provided as a computer program. We can compute using propagation of properties throughout the computation using rules similar to the chain rule for derivative propagation.

- *Compiler dependencies/Data flow analysis*: A boolean variable indicating whether an intermediate variable depends on the input variable.

- *Error analysis*: We can propagate the estimates of the computational errors incurred in the computation of derivatives, by propagating the error estimates themselves during the execution of the program. PADRE2 is a computational tool [Kub91] which can compute error estimates for the derivatives along with the derivatives themselves.

- *Interval arithmetic*: In the same spirit it is also possible to propagate the intervals for a variable throughout the program to get upper and lower bounds of $y = F(x)$ given the intervals for $x$.

- *Analytical derivatives using symbolic propagation:* We can propagate symbolically the "analytical derivatives" of continuous functions. This can be used to also propagate differential equations/dynamical systems, e.g. if $x$ satisfies the differential equation $\dot{x} = g(x, t)$, then $y = F(x)$ will satisfy a differential equation $\dot{y} = h(y, t)$ which can be propagated along with the computation of the function.

In general, the concept of overloading can be applied to compute a variety of computational objects to build interesting computational technology. The basic propagation forms just the most elementary layer of the technology, but it can be built upon and made more sophisticated just like automatic differentiation.

# Bibliography


[AMB+94] Brett M. Averick, Jorge J. Moré, Christian H. Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15:285–294, 1994.

[BBKM95] Christian H. Bischof, Ali Bouaricha, Peyvand M. Khademi, and Jorge J. Moré. Computing gradients in large-scale optimization using automatic differentiation. Preprint MCS–P488–0195, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., January 1995.

[BCC+92] Christian H. Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative code from FORTRAN programs. *Scientific Programming*, 1:11–29., 1992.

[Ben96] Jochen Benary. Parallelism in the reverse mode. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, page in this volume. SIAM, Philadelphia, Penn., 1996.

[BGHK94] Christian H. Bischof, Larry Green, Ken Haigler, and T. Knauff. Calculation of sensitivity derivatives for aircraft design using automatic differentiation. In *Proceedings of the 5th AIAA/NASA/USAF/ISSMO Symposium on Multidisciplinary Analysis and Optimization, AIAA 94-4261*, pages 73–84. American Institute of Aeronautics and Astronautics, 1994.

[Bis91] Christian Bischof. Issues in parallel automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 100–113. SIAM, Philadelphia, Penn., 1991.

[BPK95] Christian H. Bischof, Gordon Pusch, and R. Knoesel. Sensitivity analysis of the MM5 weather model using automatic differentiation. Preprint MCS–P532–0895, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., August 1995.

[Bro65] C. Broyden. A class of methods for solving nonlinear simultaneous equations. *Mathematics of Computation*, 19:577–593, 1965.

[BW97] Christian H. Bischof and Po-Ting Wu. Time-parallel computation of pseudo-adjoints for a leapfrog scheme. Preprint MCS-P639-0197, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[BWS+94] Christian H. Bischof, G. Whiffen, C. Shoemaker, Alan Carle, and A. Ross. Application of automatic differentiation to groundwater transport models. In A. Peters, editor, *Computational Methods in Water Resources X*, pages 173–182. Kluwer Academic Publishers, Dordrecht, 1994.

[CC86] Thomas F. Coleman and J. Y. Cai. The cyclic coloring problem and estimation of sparse Hessian matrices. *SIAM J. Alg. Disc. Meth.*, 7:221–235, 1986.

[CGM84] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Jacobian matrices. *ACM Trans. Math. Software*, 10(3):329–345, 1984.

[CGM85] Thomas F. Coleman, Burton S. Garbow, and Jorge J. Moré. Software for estimating sparse Hessian matrices. *ACM Trans. Math. Software*, 11(4):363–377, 1985.

[CJ97] Thomas F. Coleman and Gudbjorn Jonsson. The efficient calculation of structured gradients using automatic differentiation. *SIAM Journal on Scientific Computing*, to appear. Also appeared as Technical Report CTC97TR272, Cornell Theory Center, Cornell University, 1997.

[CM84a] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Math. Programming*, 28:243–270, 1984.

[CM84b] Thomas F. Coleman and Jorge J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. on Numerical Analysis*, 20(1):187–209, 1984.

[CMM97] J. Czyzyk, M. P. Mesnier, and Jorge J Moré. The network-enabled optimization system (neos) server. Preprint ANL/MCS–P615-1096, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[Col] Thomas F. Coleman. LSOT User Guide.

[Col94] Thomas F. Coleman. Linearly constrained optimization and projected preconditioned conjugate gradients. pages 118–122. SIAM, Philadelphia, Penn., 1994.

[CPR74] A. R. Curtis, M. J. D. Powell, and J. K. Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl.*, 13:117–119, 1974.

[CSV] Thomas F. Coleman, Fadil Santosa, and Arun Verma. Efficient calculation of Jacobian and adjoint vector products in wave propagational inverse problem using automatic differentiation. Technical report, (in preparation), Theory Center, Cornell University.

[CSV97] Thomas F. Coleman, Fadil Santosa, and Arun Verma. Semi-automatic differentiation. In *Proceedings of Optimal Design and Control Workshop.* 1997.

[CVa] Thomas F. Coleman and Arun Verma. ADMAT: An automatic differentiation toolbox for matlab. Technical report, in preparation.

[CVb] Thomas F. Coleman and Arun Verma. On preconditioning conjugate gradients for linear equality constrained minimization. Technical report, (In preparation) Theory Center, Cornell University.

[CV96a] Thomas F. Coleman and Arun Verma. Structure and efficient Hessian calculation. In Ya xiang Yuan, editor, *Advances in Nonlinear programming, proceedings of 96 International conference on nonlinear programming*, pages 57–72. Kluwer Academic Publishers, Boston/Dordrecht/London, 1996.

[CV96b] Thomas F. Coleman and Arun Verma. Structure and efficient Jacobian calculation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 149–159. SIAM, Philadelphia, Penn., 1996.

[CV97] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and matlab interface toolbox, User Guide. Technical Report CTC97TR271, Theory Center, Cornell University, 1997.

[CV98a] Thomas F. Coleman and Arun Verma. ADMIT-1: Automatic differentiation and matlab interface toolbox. 1998.

[CV98b] Thomas F. Coleman and Arun Verma. ADMIT-2: Automatic differentiation and matlab interface toolbox for structured computation, User Guide. Technical report, in preparation, 1998.

[CV98c] Thomas F. Coleman and Arun Verma. The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 19(4):1210–1233, July 1998.

[EB96] Peter Eberhard and Christian Bischof. Automatic differentiation of numerical integration algorithms. Preprint MCS–P621–1196, MCS, Argonne National Laboratory, Argonne, Ill., 1996.

[Ebe96] Peter Eberhard. The adjoint variable method for the sensitivity analysis of multibody systems interpreted as continuous, hybrid form of automatic differentiation. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, page in this volume. SIAM, Philadelphia, Penn., 1996.

[GBC+93] Andreas Griewank, Christian H. Bischof, George F. Corliss, Alan Carle, and Karen Williamson. Derivative convergence for iterative equation solvers. *Optimization Methods and Software*, 2:321 – 355, 1993.

[GC91] Andreas Griewank and George F. Corliss, editors. *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. SIAM, Philadelphia, Penn., 1991.

[GJU96] Andreas Griewank, David Juedes, and Jean Utke. ADOL–C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. On Math. Software*, 22(2):131–167, June 1996.

[Gri90] Andreas Griewank. Direct calculation of Newton steps without accumulating Jacobians. In T. F. Coleman and Yuying Li, editors, *Large-Scale Numerical Optimization*, pages 115–137. SIAM, Philadelphia, Penn., 1990.

[Gri92a] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1:35–54, 1992.

[Gri92b] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, 1(1):35–54, 1992. Also appeared as Preprint MCS–P228–0491, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., April 1991.

[Gri93] Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. In P. Pardalos, editor, *Complexity in Nonlinear Optimization*, pages 128–161. World Scientific Publishers, 1993.

[Gri94] Andreas Griewank. Tutorial on computational differentiation and optimization. In *Proceedings of XV Internationak Mathematical Programming Symposium*, Ann Arbor, MI, 1994. University of Michigan.

[GU95] Andreas Griewank and Jean Utke. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. Technical Report, TU Dresden, Institut für Wissenschaftliches Rechnen, Dresden, 1995.

[GU96] Andreas Griewank and Jean Utke. Automatic computation of sparse jacobians by applying the method of Newsam and Ramsdell. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, page in this volume. SIAM, Philadelphia, Penn., 1996.

[GUG96] Uwe Geitner, Jean Utke, and Andreas Griewank. Automatic computation of sparse jacobians by applying the method of newsam and ramsdell. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques Applications, and Tools*, page in this volume. SIAM, Philadelphia, Penn., 1996.

[HMB97] Paul Hovland, Bijan Mohammadi, and Christian H. Bischof. Automatic differentiation of navier-stokes computations. Preprint MCS-P687-0997, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., 1997.

[Hov97] Paul Hovland. *Automatic Differentiation of parallel programs*. Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, Illinois, 1997.

[Hul98] John C. Hull. *Introduction to futures and options markets*. Prentice Hall, 1998.

[Iri91] Masao Iri. History of automatic differentiation and rounding estimation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 1–16. SIAM, Philadelphia, Penn., 1991.

[JT96] Robert Jarrow and Stuart Turnbull. *Derivative Securities*. South-Western College Publications, 1996.

[Jue91] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991.

[Kub91] Koichi Kubota. PADRE2, a FORTRAN precompiler yielding error estimates and second derivatives. In Andreas Griewank and George F.

Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 251–262. SIAM, Philadelphia, Penn., 1991.

[NR83] G. N. Newsam and J. D. Ramsdell. Estimation of sparse jacobian matrices. *SIAM J. Alg. Disc. Meth.*, 4(3):404–417, 1983.

[PT79] M. J. D. Powell and Ph. L Toint. On the estimation of sparse Hessian matrices. *SIAM J. Numer. Anal.*, 16:1060–1074, 1979.

[RDG93] Nicole Rostaing, Stéphane Dalmas, and André Galligo. Automatic differentiation in Odyssée. *Tellus*, 45A:558–568, 1993.

[SH95] T. Steihaug and S. Hossain. Computing a sparse jacobian matrix by rows and columns. Technical Report, Department of Computer Science, University of Bergen, 1995.

[Shi93] Dmitri Shiriaev. *Fast Automatic Differentiation for Vector Processors and Reduction of the Spatial Complexity in a Source Translation Environment*. Ph.D. dissertation, Institute für angewandte Mathematik, Universität Karlsruhe, 1993.

[SS88] Fadil Santosa and William W Symes. Computation of the Hessian for least-squares solutions of inverse problems of reflection seismology. *Inverse problems*, 4:211–233, 1988.

[TMC+96] Anne E Trefethen, Vijay S Menon, Chi-Chao Chang, Grzegorz J. Czajkowski, Chris Myers, and Lloyd N. Trefethen. MultiMATLAB: MATLAB on multiple processors. Technical Report CTC96TR239, Cornell Theory Center, 1996.

[Tre97] Lloyd N Trefethen. *Finite Difference and Spectral Methods for Ordinary and Partial Differential Equations*. SIAM, 1997.

[ZMR88] N. Zabaras, S. Mukherjee, and O. Richmond. An analysis of inverse heat transfer problems with phase changes using an integral method. *Transactions of American Society of Mechanical Engineers(ASME)*, 110:554–561, 1988.