ON THE EFFICIENT DETERMINATION OF HESSIAN MATRIX SPARSITY PATTERN - ALGORITHMS AND DATA STRUCTURES

MARZIA SULTANA
Bachelor of Science, Military Institute of Science and Technology, 2010

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Marzia Sultana, 2016

ON THE EFFICIENT DETERMINATION OF HESSIAN MATRIX SPARSITY PATTERN - ALGORITHMS AND DATA STRUCTURES

MARZIA SULTANA

Date of Defence: August 11, 2016

Dr. Shahadat Hossain Supervisor	Professor	Ph.D.
Dr. Daya Gaur Committee Member	Professor	Ph.D.
Dr. Robert Benkoczi Committee Member	Associate Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Dedication

To

My Parents.

Abstract

Evaluation of the Hessian matrix of a scalar function is a subproblem in many numerical optimization algorithms. For large-scale problems often the Hessian matrix is sparse and structured, and it is preferable to exploit such information when available. Using symmetry in the second derivative values of the components it is possible to detect the sparsity pattern of the Hessian via products of the Hessian matrix with specially chosen direction vectors. We use graph coloring methods and employ efficient sparse data structures to implement the sparsity pattern detection algorithms.

Acknowledgments

I would like to express my full gratitude to everyone who encouraged and supported me throughout this thesis work.

First and foremost, I would like to render my utmost thanks and appreciation to my supervisor, Dr. Shahadat Hossain, for his continuous guidance, support, cooperation, and persistent encouragement throughout the journey of my MSc program. His direction, valuable opinion and effort have led me on the path of my thesis.

I wish to express my thanks to my supervisory committee members, Dr. Daya Gaur and Dr. Robert Benkoczi for there encouragement and insightful advice.

I am grateful to my parents, friends and fellow graduate students for their inspiration and support to made this path easier for me. Finally, I would like to give a special thanks to my sister for her unwavering encouragement.

Contents

List of Tables

List of Figures

List of Symbols

The next list describes several symbols that will be later used within the body of this thesis.

H, A, S	An uppercase letter is used to denote a matrix.
$A^\top$	Transpose of a matrix.
$\mathcal{F}$	The *flaw* matrix.
Φ	Structurally orthogonal column partition of a matrix.
$a_{ij}, A(i,j)$	The (i,j) entry of A matrix.
$A(i,:)$	The i^{th} row of H matrix for $A \in \mathbb{R}^{m\times n}$.
$A(:,j)$	The j^{th} column of H matrix For $A \in \mathbb{R}^{m\times n}$.
$\mathcal{S}(A)$	Sparsity pattern of matrix A.
nnz	Number of nonzero elements in a matrix.
ρ_i	Number of nonzeroes in row i of a matrix.
ρ_{max}	Maximum number of nonzeroes in any row.
$(.)$	A zero entry in matrix and any other symbol in a matrix denotes a nonzero entry.
$v(i)$	The i^{th} element of vector $v \in \mathbb{R}^n$.
$A(:,v)$	Submatrix comprised of columns whose indices are contained in vector v.
$A(u,:)$	Submatrix comprised of rows whose indices are contained in vector u.

Chapter 1

Introduction

In many scientific computing algorithms, repeated evaluation of Jacobian or Hessian matrices is a common subproblem. With a prior known sparsity and structure information, storage of and computation with the matrix can be made highly efficient. Exploitation of sparsity and structure is especially critical for applications such as large-scale scientific simulations (CM5 Weather Model, circuit simulation, finite-element methods etc) where a single simulation run can take hours on modern super computers. Taking the effective advantage of the structure of a sparse matrix requires a combination of numerical and combinatorial methods (graph algorithms and related data structures).

1.1 Motivation

Matrix sparsity can be exploited in two different ways. First, elements that are known to be zero are not stored explicitly. This can yield significant saving in computer storage of data. For example, the *WWW* webgraph is established to have 3.5 billion webpages and 128.7 billion hyperlinks resulting the graph being extremely sparse. Secondly, operations involving known zero are avoided. On the other hand, sparse matrix computations are more challenging than their dense counterparts. The computational complexity of sparse matrix operations is affected by factors such as memory traffic, size, and organization of cache memory, in addition to the number of floating point operations.

In this thesis, we investigate the problem of detection of sparsity pattern of large sparse Hessian matrices. There are efficient computational methods that exploit known sparsity to determine sparse Hessian matrices [34, 18, 36, 21, 39]. Many of these determination methods partition the columns into groups such that the nonzero unknowns in each group can be determined uniquely by a matrix-vector product of the form Hd, where H is the matrix to be determined and d is a direction determined by the columns in a group.

For a once continuously differentiable function $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ in some neighborhood of $x \in \mathbb{R}^n$ we can write

$$\left.\frac{\partial F(x+ts)}{\partial t}\right|_{t=0} = F'(x)s \equiv As \approx \frac{1}{\varepsilon}[F(x+\varepsilon s) - F(x)] \equiv b, \tag{1.1}$$

where $\varepsilon > 0$ is a small increment and $s \in \mathbb{R}^n$ is a given direction. By taking s to be unit coordinate vectors $e_i, i = 1, \ldots, n$ the nonzero elements of $F'(x)$ can be approximated with n extra function evaluations (assuming that $F(x)$ has been evaluated already). In this thesis we are concerned with the detection of zero-nonzero structure of symmetric Hessian matrices $F'(x)$ i.e., where F is the gradient of a twice continuously differentiable function $f : \mathbb{R}^n \mapsto \mathbb{R}$. Great savings in computation can be achieved if the sparsity structure of the Hessian is known a priori and remain constant in the region of interest. Unfortunately, determining the sparsity structure by hand, even when the source code for the function evaluation program is available, can be tedious and highly error-prone. Algorithmic Differentiation (AD) tools such as ADOL-C [37], ADMAT [1], MAD [16] provide facilities for sparsity calculation from the given function evaluation code. These AD tools use the underlying computational graph to accumulate and propagate the dependencies that exist between intermediate quantities. In essence, the sparsity information is obtained as the bipartite graph that results from the computational graph. If the source code for function evaluation is unavailable or if the function is provided as a black-box (as in numerical simulation) alternative techniques are needed. Griewank and Mitev [22] propose a probing method to determine the sparsity pattern of a Jacobian matrix using AD. Walther [35] proposed a new algorithm for obtaining Hessian sparsity patterns which is essentially a forward mode AD procedure and was implemented in ADOL-C.

1.2 Our Contribution

In this thesis, we assume that the Hessian matrix is available only as a black box in that given a direction d as input, an approximation to the Hessian times the direction d can be obtained at a cost proportional to an extra gradient evaluation. Further, we assume that there is an evaluation procedure for gradients at a given point. Using symmetry in the second derivative values of the components it is possible to determine the sparsity pattern of the Hessian via products of the Hessian matrix with specially chosen direction vectors. We employ graph coloring methods and give efficient sparse data structures to implement the sparsity pattern detection algorithms. Results from numerical testing confirm the effectiveness of our method. Specific contributions are given below.

1. We propose sparse data structures, extend the multilevel algorithm of [7], and present an efficient implementation of Hessian matrix sparsity detection.

2. We employ ordering methods [24] to improve column coloring over the greedy method suggested by Carter in [7].

3. The seed matrix and the compressed Hessian are expanded to full matrix to identify "*flawed* entries" in [7]. Our implementation uses sparse data structures and avoids expansion to full matrix.

4. The set of possibilities for the identification of possible flawed entries are only implicitly generated; the pattern matrix is never generated explicitly. This is in contrast to the proposal in [7].

5. We present results of numerical testing using a suite of large-scale practical problem instance to validate our approach.

Parts of the work contained in this thesis,

- has been accepted to appear in the ACM Communications in Computer Algebra (CCA) [9].

- will be presented as a refereed poster paper at the 41st International Symposium on Symbolic and Algebraic Computation (ISSAC) to be held in Waterloo, Ontario, July 19-22, 2016 [8].
- was presented as a refereed poster paper at the workshop on Nonlinear Optimization Algorithms and Industrial Applications that celebrated the 70th birthday of Andrew R. Conn, held in the Fields Institute, Toronto, Ontario, June 2-4, 2016 [30].

1.3 Thesis organization

Including this chapter, there are five more chapters in this thesis organized as follows.

In Chapter 2, we review some basic concepts used in this thesis and a finite difference approximation scheme that is suitable for exploiting known sparsity is introduced. For completeness we also describe Algorithmic Differentiation (AD), which can easily apply to our method to avoid truncation error.

In Chapter 3, we describe efficient data structures to store a sparse matrix in computer memory and extend the same to enable operations on sparse matrices that we have used for our algorithms.

In Chapter 4, we describe the voting scheme to determine the missing pattern elements and the multilevel algorithm to determine the sparsity pattern of Hessian matrix.

In Chapter 5, we provide experimental results that demonstrate the efficacy of our algorithms.

Finally in Chapter 6, we provide concluding remarks and directions for future research in this area.

Chapter 2

Problem Definition and Background Study

In this chapter, we introduce our problem more technically and review mathematical preliminaries and definitions necessary for this thesis. We also discuss about the partitioning algorithms to determine the sparse Hessian matrices.

2.1 Problem Definition

To introduce the problem considered in this thesis, we consider the unconstrained minimization problem

$$\min_{x \epsilon \mathbb{R}^n} f(x)$$

Assuming that $f(x)$ is twice continuously differentiable, let the first derivative (the gradient vector) of $f(x)$ be denoted by $g(x) = \nabla f(x)$ and the second derivative (the Hessian matrix) of $f(x)$ be denoted by $H(x) = \nabla^2 f(x)$. A quadratic approximation to the nonlinear function $f(x)$ can be written as,

$$q(x) = x_0 + g^T x + \frac{1}{2} x^T H x \tag{2.1}$$

where, x_0 is a constant. The unique minimizer to the quadratic $q(x)$ can be found by taking the derivative of $q(x)$ with respect to x and solving the resulting equation,

$$\nabla f(x) = g + Hx = 0 \tag{2.2}$$

Rewriting equation (2.2),

$$Hx = -g \tag{2.3}$$

which is a system of linear equations that can be solved for the unknown vector x. Equation (2.3) is commonly called the Newton equation. The key idea is to use the solution to the linear system to obtain a 'step' such that a new approximate 'solution' to the original minimization problem is "better" than the current approximation. This observation leads to the following algorithm for Newton's method.

Algorithm: Newton's Method

```
   Input: Given an initial approximation $x_0$ and a convergence tolerance $t_0$
1  for $k \leftarrow 0$ to max_iter do
2      Evaluate: $g(x_k)$ and $H(x_k)$ ;
3      Solve for $S_k$: $H(x_k)S_k = -g(x_k)$ ;
4      Update: $x_{k+1} \leftarrow x_k + S_k$ ;
5      Check for Convergence:
6      if $||g(x_{k+1})|| \leq t_0$ then
7          break;
```

One of the main advantages of Newton's method is its fast rate of convergence (quadratic under some favorable assumptions) and a well-developed convergence theory. For a detailed description of the convergence theory and variants of Newton's method we refer to [15].

One of the main difficulties in using Newton's method on practical problems is the need for derivative information at each iteration. Fortunately in many large-scale problems, the Hessian matrix is sparse or structured.

2.2 Preliminaries

In this section, we introduce some basic terminologies related to this thesis and discuss different methods for obtaining derivative information.

2.2.1 Gradient

Let $f : \mathbb{R}^n \mapsto \mathbb{R}$ be a twice continuously differentiable function. The gradient of function $f(x_1, x_2, x_3, ..., x_n)$ is denoted by,

$$\nabla f = \frac{\partial f}{\partial x_1} e_1 + \cdots + \frac{\partial f}{\partial x_n} e_n = \begin{bmatrix} \dfrac{\partial f}{\partial x_1} \\ \vdots \\ \dfrac{\partial f}{\partial x_n} \end{bmatrix} \tag{2.4}$$

whose components $\frac{\partial f}{\partial x_i}$ are the partial derivatives of f and the e_i are the orthogonal unit vectors pointing in the coordinate directions.

2.2.2 Hessian

The Hessian matrix is a square matrix of second order partial derivatives of a scalar-valued function f.

$$H = \begin{bmatrix} \dfrac{\partial f}{\partial x_1}\left(\dfrac{\partial f}{\partial x_1}\right) & \cdots & \dfrac{\partial f}{\partial x_1}\left(\dfrac{\partial f}{\partial x_n}\right) \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f}{\partial x_n}\left(\dfrac{\partial f}{\partial x_1}\right) & \cdots & \dfrac{\partial f}{\partial x_n}\left(\dfrac{\partial f}{\partial x_n}\right) \end{bmatrix} = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x_1^2} & \cdots & \dfrac{\partial^2 f}{\partial x_1 x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial^2 f}{\partial x_n x_1} & \cdots & \dfrac{\partial^2 f}{\partial x_n^2} \end{bmatrix} \tag{2.5}$$

The second-derivative is independent of the order in which derivatives are taken. Hence, $H(i, j) = H(j, i)$ for every pair (i, j) and the Hessian is a symmetric matrix.
For example, let $f(x_1, x_2) = x_1^3 + x_1^2 x_2 - x_2^2 - 4x_2$
The gradient is,

$$\nabla f(x_1, x_2) = \begin{bmatrix} 3x_1^2 + 2x_1 x_2 \\ x_1^2 - 2x_2 - 4 \end{bmatrix}$$

The Hessian matrix is

$$H(x_1,x_2) = \begin{bmatrix} 6x_1+2x_2 & 2x_1 \\ 2x_1 & -2 \end{bmatrix}$$

Definition 2.1. The *Sparsity Pattern* or *Sparsity Structure* of matrix $A \in \mathbb{R}^{m\times n}$ is a specification of A's nonzero entries.

In this thesis matrix A's sparsity pattern is denoted as $\mathcal{S}(A) = \{(i,j) \mid a_{ij} \neq 0, i,j = 1,2,\ldots,n\}$ where $A \in \mathbb{R}^{n\times n}$. A sparsity pattern is symmetric if (j,i) is in the sparsity pattern whenever (i,j) is in the sparsity pattern. The entries of the matrix outside the sparsity pattern are allowed to be zero.

The problem of determination of a Hessian matrix H can be formulated as follows:

- Given the sparsity structure of a symmetric matrix H of order n.
- Obtain vectors $d_1,d_2,\ldots,d_p$ such that $Hd_1,Hd_2,...,Hd_p$ and determine H uniquely.

We will assume that the diagonal elements of H are non zero (since in a minimization problem the Hessian is usually positive definite at a minimizer). The interesting part is obtaining difference vectors $d_1,d_2,...,d_p$ with p as small as possible, since the evaluation of the gradient can be expensive.

Several methods have been proposed to estimate the derivative information of a Hessian matrix. Curtis, Powell, and Reid [14] observed that the elements of H can be determined directly if the directions partition the columns into groups such that columns in the same group do not have a nonzero element in the same row position. Coleman and More [12] used the partition problem with a certain graph coloring problem and their numerical results show that the improved algorithms are nearly optimal or optimal for practical problems. They also showed that at least ρ_{max} directions are required to determine a general matrix H uniquely, where ρ_{max} is the maximum number of non zeroes in any row. Powell and Toint [34] were the first to show that exploiting symmetry can result in significant benefits for many sparsity structures and proposed several algorithm which exploits symmetry. Automatic differentiation (AD) can be used to accurately determine these matrices and can also determine the sparsity pattern of these matrices. In order to obtain the entire Hessian matrix, the graph coloring techniques explore sparsity patterns of the matrix and also reduce the cost of matrix-vector products.

2.2.3 Finite Difference Approximation

Finite differencing is an approach to the calculation of approximate derivatives whose motivation comes from Taylor's theorem. For instance, we can get an approximation of the partial derivative of f by evaluating the objective with respect to the i^{th} variable x_i using the forward difference formula,

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x+\varepsilon e_i) - f(x)}{\varepsilon} \tag{2.6}$$

where $i = 1, 2, ..., n$, e_i is the i^{th} unit vector that is, the vector whose elements are all 0 except for a 1 in the i^{th} position and $\varepsilon > 0$ is small.

2.2.4 Approximating the Hessian

It is possible to obtain the Hessian matrix by using finite difference approximation. By using the graph coloring techniques sparse Hessians often can be approximated in this manner by using considerably fewer than n direction vectors. Many important algorithms do not require knowledge of the full Hessian, instead each iteration requires the Hessian-vector product $\nabla^2 f(x)p$, for a given vector p. We can obtain an approximation to this matrix-vector product by Taylor's theorem. When second derivatives of f exist and are Lipschitz continuous near x, we have

$$\nabla^2 f(x)p \approx \frac{\nabla f(x+\varepsilon p) - \nabla f(x)}{\varepsilon} \tag{2.7}$$

For the case in which even gradients are not available, we can use Taylor's theorem once again for approximating the Hessian that use only function values [33]. When the Hessian is sparse the key observation is that, because of symmetry, any estimate of the element $\nabla^2 f(x_{i,j}) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j}$ is also an estimate of its symmetric counter part $\nabla^2 f(x_{j,i})$. By exploiting symmetry, it is possible to estimate the entire Hessian by evaluating ∇f and using the formula 2.7.

2.2.5 Algorithmic Differentiation

Algorithmic Differentiation (AD), also called computational differentiation is a novel way to differentiate a function f to compute derivative matrices, where f is given by a computer program. AD uses exact formulas along with floating-point values and it involves no approximation error as in finite difference. This is done in such a way that the function computed by the computer program is simply a composition of these elementary functions $(\sin, \cos, \exp, \log, ...$ denoted by $\Phi)$ and operations $(+, -, x/, ...)$.
For a function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ the evaluation procedure is a sequence of scalar assignments,

$$v_{i-n} = x_i \text{ or } v_i = \Phi_i(v_j) \text{ for } i = 1, ..., n$$

where, x_i are independent variables, v_i are internal variables and obtain values v_i by applying elemental function Φ_i to some set of arguments $v_j (j < i)$. Finally the output variables for the function are extracted from the final intermediate variables.
As an example, consider the function $f = x_1^2 + \sin(x_3) - (2x_2)^2$
Here the graph on above figure called computational graph (a directed acyclic graph to visualize the evaluation procedure of AD). The sequence of elementary operations is known as code list, where x_1, x_2 and x_3 are independent variables, $v_i = -2, -1, ..., 6$ are called the adjoint variables and f is the output variable [23]. It is possible to generate different code list for the same function. After forming a code list, we can apply rules of differentiation to compute the derivative of function f with respect to the independent variables (x_1, x_2 and x_3).
Usually, two distinct modes of AD are presented, forward accumulation (or forward mode) and reverse accumulation (or reverse mode).

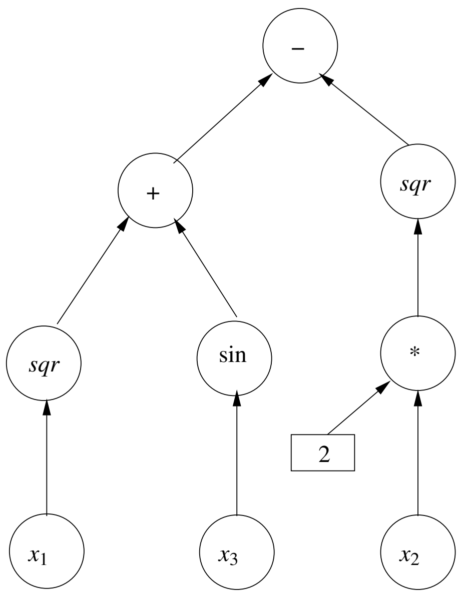

Figure 2.1: Evaluation Procedure of function f in Computational Graph

Table 2.1: Code List of function f

$$v_{-2} = x_1$$

$$v_{-1} = x_3$$

$$v_0 = x_2$$

$$v_1 = v_{-2}^2$$

$$v_2 = \sin(v_{-1})$$

$$v_3 = v_0 * 2$$

$$v_4 = v_3^2$$

$$v_5 = v_1 + v_2$$

$$v_6 = v_5 - v_4$$

$$f = v_6$$

2.2.6 Forward Accumulation

In forward accumulation AD, we need to identify the independent variable to which differentiation is performed and computed the derivative of each sub-expression recursively.

One simply augments each variable v with its derivative $\dot{v} = \frac{\partial v}{\partial x}$. For example, we are calculating the derivative of above function f with respect to independent variable x_1 in forward mode. The function evaluation procedure for forward mode is,

Table 2.2: Evaluation Procedure for Forward Mode

$v_{i-n} = x_i$	*for* i= 1,...,n
$v_i = \Phi_i(v_j)$	*where* $j < i$ *for* i=1,...,l
$f_{m-i} = v_{l-i}$	*for* i= m-1,...,0

$$\dot{v_{-2}} = 1$$

$$\dot{v_{-1}} = 0$$

$$\dot{v_0} = 0$$

$$\dot{v_1} = 2 * v_{-2}$$

$$\dot{v_2} = 0$$

$$\dot{v_3} = 0$$

$$\dot{v_4} = 0$$

$$\dot{v_5} = 2 * v_1 + 0$$

$$\dot{v_6} = 2 * v_1 - 0$$

$$f = 2x_1$$

As the derivatives are then computed in the evaluation steps in forward accumulation and combined with other derivatives via the chain rule, the computational complexity of one pass is proportional to the complexity of the original code.

2.2.7 Reverse Accumulation

In reverse accumulation AD, we need the dependent variable to be differentiated and computes the derivative with respect to each sub-expression recursively by using the chain rule. Derivative of a chosen dependent variable f with respect to a sub expression v is, $\overline{v} = \frac{\partial f}{\partial v}$. For example, the function evaluation procedure of calculating the derivative of above function f in reverse mode is,

Table 2.3: Evaluation Procedure for Reverse Mode

$v_{l-i} = f_{m-i}$	*for* i= 0,...,m-1
$v_j = v_i \Phi_i(v_i)$	*where* $j < i$, *for* i=l,...,1
$x_i = v_{i-n}$	*for* i= n,...,1

$$f = v_6$$

$$\overline{v_6} = 1$$

$$\overline{v_5} = \overline{v_6} = 1$$

$$\overline{v_4} = -\overline{v_6} = -1$$

$$\overline{v_3} = 2 * v_3 * \overline{v_4} = -4x_2$$

$$\overline{v_2} = \overline{v_5} = 1$$

$$\overline{v_1} = \overline{v_5} = 1$$

$$\overline{v_0} = 2 * \overline{v_3} = -8x_2$$

$$\overline{v_{-1}} = \cos(v_{-1}) * \overline{v_2} = \cos(x_3)$$

$$\overline{v_{-2}} = 2 * v_{-2} * \overline{v_1} = 2x_1$$

The accumulation of adjoint derivatives are computed in reverse order as the function values are computed. Forward accumulation is more efficient than reverse accumulation for functions $f : R_n \to R_m$ with $m \gg n$ as only n passes are necessary, where reverse accumulation is more efficient than forward accumulation with $m \ll n$ as only m passes are necessary.

2.3 Determination of Sparse Derivative Matrices

Curtis, Powell and Reid proposed an algorithm to estimate the sparse Jacobian matrix called CPR algorithm [14], which is based on the partitioning of columns of the Jacobian. The CPR method uses a greedy technique to partition columns of a matrix A into structurally orthogonal groups. They observed that if the directions partition the columns into structurally orthogonal groups, then the elements of A can be determined directly.

Definition 2.2. *Structurally Orthogonal Partitioning* of the columns of a matrix A is the partition of the columns of A into groups in which no two columns in a group have the nonzero elements in the same row position. In other words, columns $A(:,j)$ and $A(:,k)$ are structurally orthogonal if there is no such row index i for which $a_{ij} \neq 0$ and $a_{ik} \neq 0$.

Coleman and Moré [12] further analyze the column partitioning problem and suggest partitioning algorithm based on graph coloring heuristics. They showed that the problem of finding a minimum cardinality column partitioning consistent with *direct determination* is equivalent to a vertex coloring problem of an associated graph and that the problem is NP-Hard. On the other hand, efficient coloring heuristics developed over the years have been found to yield nearly optimum coloring. Goldfarb and Toint [20] studied such optimal estimation of Jacobians and Hessians arising in the finite difference approximations of partial differential equations. Gebremedhin, Manne, and Pothen [19] presented a comprehensive overview of graph coloring methods for sparse Jacobian and Hessian matrix determination. They developed uniform graph-theoretic framework for studying the matrix partitioning problems. Powell and Toint [34] extended the CPR method to compute sparse Hessians by taking the advantage of sparsity structure and symmetry of the second derivative matrix. McCormick [32] introduced a distance-2 graph coloring model for the computation of Hessians and proposed two new ways of classifying direct methods for this problem.

2.3.1 The CPR Algorithm

If a group of columns of matrix A, say columns j and l, are *structurally orthogonal*, i.e., no two columns have nonzero entries in the same row position, only one extra function evaluation,

$$F'_j + F'_l = A(:,j) + A(:,l) \approx \frac{1}{\varepsilon}[F(x + \varepsilon(e_j + e_l)) - F(x)], \tag{2.8}$$

is sufficient to read-off the nonzero entries from the product $b = As$, with $s = e_j + e_l$. The key observation here is that if the sparsity pattern of a group of columns is such that for every pair of column indices $j \neq l$ in the group $\mathcal{S}(A(:,j)) \bigcap \mathcal{S}(A(:,l))$ is empty, then only one extra function evaluation will suffice to determine the nonzero entries in those columns. In other words, columns $A(:,j)$ and $A(:,l)$ are structurally orthogonal.

Let the sparsity pattern $\mathcal{S}(A)$ of $A \in \mathbb{R}^{n\times n}$ where $A = A^\top$ be given. Let Φ be a mapping $\Phi : \{1,2,\ldots,n\} \mapsto \{1,2,\ldots,p\}$ such that $\Phi(i) = \Phi(j)$ implies columns $A(:,i)$ and $A(:,j)$ are structurally orthogonal where p is the total number of groups in Φ, i.e. $|\Phi| = p$. With Φ and an appropriately chosen ε in Equation (2.8) for direction vectors $S(:,k), k = 1,2,\ldots,p$ the following matrix equation,

$$AS = B \tag{2.9}$$

can be solved to recover the nonzero entries. The nonzero entries are simply identified in the compressed matrix B, where matrix B is obtained via FD (or AD forward accumulation). Thus, the columns of matrix B correspond to the directional derivatives of function F in the directions $S(:,k), j = 1,\ldots,p$; with FD each directional derivative costs one extra evaluation of function F while with AD it is a small multiple (usually 2-3) of the cost of evaluating the function F [29]. It is important to notice that in a CPR-compression the number of nonzero elements in any row is preserved. Then to determine the nonzero entries uniquely at least ρ_{max} products are needed in any method based on matrix-vector product calculation, where ρ_{max} is the maximum number of nonzero elements in any row of A [26]. The following algorithm can be used to compute matrix B.

Algorithm 1: Computing B matrix

```
   Input  : The matrix $A$
            function $\Phi$
   Output: Matrix $B = AS$ , where $S \in \mathbb{R}^{n \times p}$
1  for $k$=1 to $p$ do
2      construct $S(:k)$ ;
3      compute $B(:,k) \leftarrow AS(:,k)$ ;
```

We illustrate Algorithm 1 with a small example. Let the matrix A in Figure (2.2) be the Hessian matrix of some function F at x. There are three structurally orthogonal column

$$A = \begin{bmatrix} a_{11} & \cdot & a_{13} & \cdot & \cdot \\ \cdot & a_{22} & \cdot & \cdot & a_{25} \\ a_{31} & \cdot & a_{33} & \cdot & a_{35} \\ \cdot & \cdot & \cdot & a_{44} & \cdot \\ \cdot & a_{52} & a_{53} & \cdot & a_{55} \end{bmatrix}$$

$$\Phi \rightarrow \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 2 & 1 & 3 \\ \hline \end{array}$$

Figure 2.2: Matrix A with its structurally orthogonal mapping Φ.

groups in matrix A, i.e. $p = 3$. Therefore matrix A can be approximated with three extra function evaluations of the form in Equation (2.8) for direction set to $e_1 + e_2 + e_4$, e_3 and e_5 in addition to evaluating F at x. The nonzero entries of matrix A can be determined using Equation (2.9) with,

$$S = [e_1 + e_2 + e_4 \quad e_3 \quad e_5] \text{ and}$$

$$B = \begin{bmatrix} a'_{11} & a'_{13} & \cdot \\ a'_{22} & \cdot & a'_{25} \\ a'_{31} & a'_{33} & a'_{35} \\ a'_{44} & \cdot & \\ a'_{52} & a'_{53} & a'_{55} \end{bmatrix}$$

Where $(')$ indicates that the entry in matrix B is an small approximation to the true value. The values in matrix B are computed by using AD forward mode, thus free of truncation error and are obtained at a small constant multiple of the cost of evaluating the function F [23]. The columns of matrix B correspond to the directional derivatives of function F in the directions $S(:,j), j = 1,2$ and 3. For a given index pairs (i,j), $A(i,j)$ will be found in $B(i,k)$ where $k = \Phi(j)$. It can be verified that the partitioning Φ is *optimal* for the given example, which means every other partition Φ must have at least 3 groups.

Chapter 3

Flaw Computation

In this chapter, we describe the efficient data structures to store a sparse matrix in computer memory and a method for the detection of mislabelled entries in sparsity patterns. We also introduce an efficient sparse data structure for *'flaw'* calculations.

3.1 Sparse Matrix

A definition of Sparse matrix by Wilkinson [38] is that, a matrix is called sparse if it is computationally advantageous to utilize the knowledge that many of its entries are zero. We can define sparsity as the fraction of nonzero elements over the total number of elements.

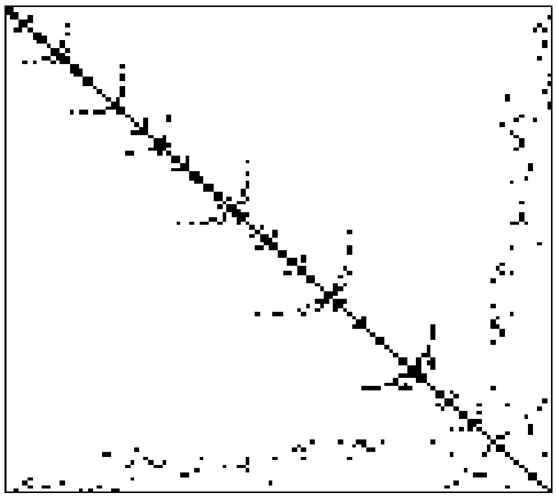

Figure 3.1: Example of Sparse Matrix (Power Network Pattern, the non-zero elements are shown in black) Name: *bcspwr03*, Dimensions: 118×118, 476 nonzero elements, Source: [4]

3.2 Data Structure for Sparse Matrix

It is beneficial to use specialized algorithms and data structures to take advantage of sparse structure for storing and manipulating sparse matrices on a computer. A matrix is usually stored using two-dimensional array but it is slow and inefficient for large sparse matrices as the processing and memory are wasted on the zeroes. Sparse data is by nature more easily compressed and thus require significantly less storage.
For example, the *bcsstk0* matrix in Figure (3.2) is a small test problem of dimension 48×48 from BCS Structural Engineering matrices.

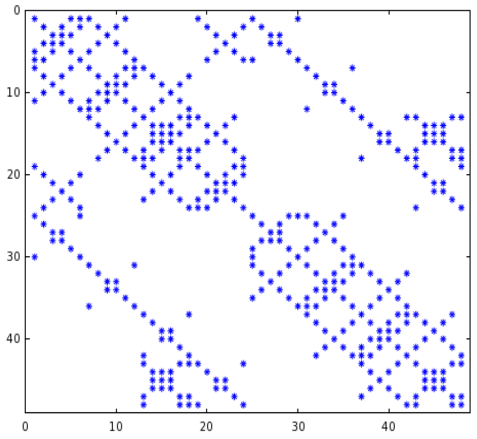

Figure 3.2: Sparse Matrix, Name: *bcsstk01*, Dimensions: 48×48, 400 nonzero elements are shown in blue, Source: [2]

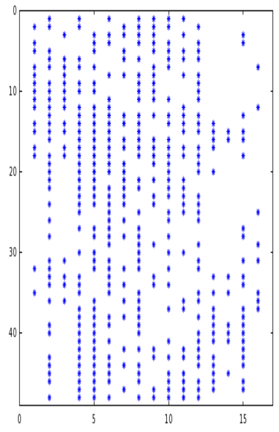

Figure 3.3: Compressed Matrix *bcsstk01*, Dimention: 48×16

There are 16 structurally orthogonal column groups in a structurally orthogonal partition of matrix *bcsstk0*. So it is possible to compress the matrix columns into 16 from 48. The matrix in Figure (3.3) is a compressed form of matrix *bcsstk01* where the dimension is now 48×16.

Depending on the number and distribution of the nonzero entries, different data structures can be used. We are discussing the following data structures:

3.2.1 Coordinate Storage

In coordinate storage format the sparse matrix A is stored as a collection of 3-tuples (i, j, a_{ij}) which representing only nonzero entries, where i denotes the row index, j the column index and a_{ij} the nonzero entry. Assume we have a matrix $A^{m \times n}$ with nnz number of nonzero elements. The corresponding coordinate storage data structure of matrix A is presented in Figure (3.4),

$$A = \begin{bmatrix} a_{11} & \cdot & a_{13} & \cdot & \cdot & \cdot \\ a_{21} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & a_{32} & \cdot & \cdot & \cdot & a_{36} \\ \cdot & \cdot & \cdot & a_{44} & \cdot & \cdot \\ \cdot & \cdot & a_{53} & \cdot & \cdot & a_{56} \\ \cdot & a_{62} & \cdot & \cdot & a_{65} & \cdot \end{bmatrix}$$

value

a_{11}	a_{21}	a_{32}	a_{36}	a_{13}	a_{44}	a_{53}	a_{62}	a_{65}	a_{56}

row

1	2	3	3	1	4	5	6	6	5

col

1	1	2	6	3	4	3	2	5	6

Figure 3.4: Coordinate storage data structure of sparse matrix

where the nonzero values are stored in the memory with no specific order, so accessing the matrix elements by row or column is equally costly. The coordinate storage requires $nnz + nnz + nnz = 3nnz(H)$ units of storage.

3.2.2 Compressed Storage

The compressed Row Storage (CRS) or Compressed Column Storage (CCS) format represents a matrix by three one dimensional arrays, that contain nonzero values, the extents of rows, and column indices. The CRS format allows fast row access and matrix-vector multiplications, thus a suitable choice for our method.

The storage format of sparse matrix we used in the thesis is Compressed Row Storage. CRS uses array *row_ptr* to represent starting index of each row and array *col_ind* to store column indices of each row. Column indices of each row i can be found in between $col\_ind[row\_ptr[i]]$ and $col\_ind[row\_ptr[i+1]-1]$. CRS uses $2nnz(A)+n+1$ memory locations. For example, consider the matrix A and corresponding CRS data structure in the following Figure (3.5).

$$A = \begin{bmatrix} a_{11} & \cdot & a_{13} & \cdot & \cdot & \cdot \\ a_{21} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & a_{32} & \cdot & \cdot & \cdot & a_{36} \\ \cdot & \cdot & \cdot & a_{44} & \cdot & \cdot \\ \cdot & \cdot & a_{53} & \cdot & \cdot & a_{56} \\ \cdot & a_{62} & \cdot & \cdot & a_{65} & \cdot \end{bmatrix}$$

value

a_{11}	a_{13}	a_{21}	a_{32}	a_{36}	a_{44}	a_{53}	a_{56}	a_{62}	a_{65}

col_ind

1	3	1	2	6	4	3	6	2	5

row_ptr

1	3	4	6	7	9	11

Figure 3.5: CRS data structure of sparse matrix

CCS is similar to CSR except that values are read first by column, a row index is stored for each value in array *row_ind*, and column pointers are stored in *col_ptr*. Row indices of each column j can be found in between $row\_ind[col\_ptr[j]]$ and $row\_ind[col\_ptr[j+1]-1]$.

Hence CCS uses $2nnz(H)+m+1$ memory locations. The following Figure (3.6) shows the CCS data structure of above matrix A.

row_ind

1	2	3	6	1	5	4	6	3	5

col_ptr

1	3	5	7	8	9	11

Figure 3.6: CCS data structure of sparse matrix

3.2.3 Cache Complexity of Sparse Matrix Access

In addition to the number of floating point operations, the computational complexity of sparse matrix operations are dependent on memory traffic and the size and organization of cache memory. Cash memories are used to improve the performance by minimizing data movement between main memory and CPU (Central Processing Unit). In general, the closer the cache to the CPU, the faster the time to access and transfer data to the CPU. The cache keeps a copy of most frequently used data and supplies them to the CPU whenever requested. When a piece of data is requested by CPU, the cache memory is searched by level. A hit in cache happens if the CPU gets the requested data from the cache. Otherwise it is called a miss. The effectiveness of a cache system depends on hit rate and miss rate which are measured by the ratio of total number of hits or misses and the total number of accesses in cache. An empty cache is refereed as *cold* cache and misses in empty cache are called *compulsory* misses.

The access pattern of data in a hierarchical memory computing system is called locality of reference. The principle of data *locality* states that, recently accessed data (temporal) and sequential data (spatial) are likely to be accessed in the near future. When a miss occurs, a data block of consecutive memory locations from main memory or another cache is copied into the cache which follows the principle of spatial locality. In general, the better spatial locality reduces the cache misses and improves the cache performance [6].

If the elements are accessed in arbitrary order for each row of the sparse matrix A, then the number of cache misses can be as high as $O(nnz)$. The elements in sparse matrix in CRS or CCS format are stored in order of increasing index. Therefore, if the elements in each row in CRS (or column in CCS) of a sparse matrix A are accessed in the order they are stored in the data structure, then it is expected to achieve full spatial locality in accessing matrix A. So only the *compulsory* misses can happen caused by the first reference to a location in memory.

3.3 Detection of Missing Elements

In this section, to identify the missing nonzero elements of a Hessian matrix we,

- describe a procedure for locating missing nonzero elements with respect to a given trial pattern,
- define the notion of *"flaw"*, and
- propose data structures to implement *flaw* localization and storage.

For ease of explanation we sketch the overall algorithm for sparsity detection and use an example to illustrate the main points. Our main assumptions are, (a) the gradient evaluation is available as a black box and (b) an analytical expression for the diagonal of Hessian is available (this condition is not strictly necessary and will be relaxed later). let $\mathcal{S}(A)$ denote the true sparsity pattern of a Hessian matrix A.

```
Algorithm: An algorithm for sparsity detection
  Input: Guess an initial pattern $\mathcal{S}_0$ for $A$.
1 while true sparsity pattern $\mathcal{S}(A)$ is not determined do
2     Compute a structurally orthogonal partition $\Phi$ of the guessed pattern $\mathcal{S}_0$ ;
3     Define the direction matrix $S$ ;
4     Compute the compressed matrix $B_0$ using Algorithm 1 ;
5     Identify the possible flaw locations using value symmetry ;
6     Calculate the flaw matrix $\mathcal{F}$ using pattern symmetry ;
7     Update sparsity pattern of $\mathcal{S}_0$ ;
```

Example: Determination of Sparsity Pattern with a guess pattern $\mathcal{S}_0$

To illustrate the above algorithm consider the following example of a 10×10 Sparse Hessian matrix A in Figure (3.7). $\mathcal{S}_0$ is a initial guess pattern of A and partitioned into three column coloring groups 1, 2 and 3 [R(ed), B(lue) and G(reen)]. The elements in position $(1,5)$, $(5,1)$ and $(3,9)$, $(9,3)$ (X and Z) of matrix A are missing in the guess pattern $\mathcal{S}_0$.

$$A = \begin{bmatrix}
R & . & G & . & X & . & . & . & . & . \\
. & B & . & R & . & . & . & . & . & . \\
R & . & G & . & B & . & . & . & Z & . \\
. & B & . & R & . & G & . & . & . & . \\
X & . & G & . & B & . & R & . & . & . \\
. & . & . & R & . & G & . & B & . & . \\
. & . & . & . & B & . & R & . & G & . \\
. & . & . & . & . & G & . & B & . & R \\
. & . & Z & . & . & . & R & . & G & . \\
. & . & . & . & . & . & . & B & . & R
\end{bmatrix}$$

$$\mathcal{S}_0 = \begin{bmatrix}
R & . & G & . & . & . & . & . & . & . \\
. & B & . & R & . & . & . & . & . & . \\
R & . & G & . & B & . & . & . & . & . \\
. & B & . & R & . & G & . & . & . & . \\
. & . & G & . & B & . & R & . & . & . \\
. & . & . & R & . & G & . & B & . & . \\
. & . & . & . & B & . & R & . & R & . \\
. & . & . & . & . & G & . & B & . & R \\
. & . & . & . & . & . & R & . & G & . \\
. & . & . & . & . & . & . & B & . & R
\end{bmatrix}$$

Figure 3.7: Hessian matrix A and a guessed pattern $\mathcal{S}_0$

A structurally orthogonal column partition Φ of guessed pattern $\mathcal{S}_0$ is,

$$\Phi \rightarrow \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 1 & 2 & 3 & 1 & 2 & 3 & 1 \\ \hline \end{array}$$

The direction Matrix S can be defined as,

$$S(j,k) = \begin{cases} \delta_j \neq 0, & \text{if } \Phi(j) = k \\ 0 & \text{otherwise} \end{cases}, \quad k = 1, \ldots, p$$

Where δ is an array of size n where, $\delta_i > 0$ is small and represents ε in Equation (2.8).

$$\begin{bmatrix} \delta_1 & . & . \\ . & \delta_2 & . \\ . & . & \delta_3 \\ \delta_4 & . & . \\ . & \delta_5 & . \\ . & . & \delta_6 \\ \delta_7 & . & . \\ . & \delta_8 & . \\ . & . & \delta_9 \\ \delta_{10} & . & . \end{bmatrix}$$

Figure 3.8: Matrix S

Here we never construct matrix S explicitly. We construct one column of S at a time and compute the product AS inside Algorithm 1 (described in chapter 2) for computing matrix B. Constructing S can be accomplished as follows where s is a vector represents one column of S.

Algorithm 1.1: Constructing $S(:,k) = s$

```
Input: Array $\Phi$, that gives the partitioning
       Array $\delta$ of size $n$
1 let $s$ be the vector of all zeros ;
2 for each $j$ where $\Phi(j) = k$ do
3     $s(j) = \delta(k)$ ;
```

If we apply CPR algorithm using trial pattern $\mathcal{S}_0$ by Equation (2.9) then we have matrix B as in Figure (3.9). In many cases B will be dense i.e. most of its entries are not identically zero. So it is reasonable to use a two dimensional array for B. This will allows us to locate entry $a_{i,j}$ (or $\delta_i a_{j,i}$) in B efficiently. We can express Equation 2.9 as,

$$a_{ij}\delta_j = b_{ik} \tag{3.1}$$

$$
\begin{bmatrix}
R & . & G & . & X & . & . & . & . & . \\
. & B & . & R & . & . & . & . & . & . \\
R & . & G & . & B & . & . & . & Z & . \\
. & B & . & R & . & G & . & . & . & . \\
X & . & G & . & B & . & R & . & . & . \\
. & . & . & R & . & G & . & B & . & . \\
. & . & . & . & B & . & R & . & G & . \\
. & . & . & . & . & G & . & B & . & R \\
. & . & Z & . & . & . & R & . & G & . \\
. & . & . & . & . & . & . & B & . & R
\end{bmatrix}
\begin{bmatrix}
\delta_1 & . & . \\
. & \delta_2 & . \\
. & . & \delta_3 \\
\delta_4 & . & . \\
. & \delta_5 & . \\
. & . & \delta_6 \\
\delta_7 & . & . \\
. & \delta_8 & . \\
. & . & \delta_9 \\
\delta_{10} & . & .
\end{bmatrix}
=
\begin{bmatrix}
R*\delta_1 & X*\delta_5 & G*\delta_3 \\
R*\delta_4 & B*\delta_2 & . \\
R*\delta_1 & B*\delta_5 & G*\delta_3+Z*\delta_9 \\
R*\delta_4 & B*\delta_2 & G*\delta_6 \\
R*\delta_7+X*\delta_1 & B*\delta_5 & G*\delta_3 \\
R*\delta_4 & B*\delta_8 & G*\delta_6 \\
R*\delta_7 & B*\delta_5 & G*\delta_9 \\
R*\delta_{10} & B*\delta_8 & G*\delta_6 \\
R*\delta_7 & . & G*\delta_9+Z*\delta_3 \\
R*\delta_{10} & B*\delta_8 & .
\end{bmatrix}
$$

Figure 3.9: Compressed matrix $B = AS$

Now we can compute the missing entries in A by using value symmetry and the sparsity structure of $\mathcal{S}_0$ and matrix B. Let A_0 be the Hessian approximation by the CPR method which have "incorrect" entries.

Definition 3.1. If $A_0(i,j) \neq A_0(j,i)$, then the element at location (i,j) is called *type* 1 *flaw*.

Definition 3.2. If $A_0(i,j) = 0$ but $B(i,l) \neq 0$ where $\Phi(j) = l$, then the element at location (i,l) of matrix B is called *type* 2 *flaw*.

"*type* 1" and "*type* 2" flaws are what Carter calls "*H* flaws" and "*Y* flaws", respectively. To avoid ambiguity we do not associate the flaws name to specific matrix.

3.3.1 Determination of Flaw Locations

Let $\mathcal{S}_0 \approx \mathcal{S}(A)$ be an *approximation* to the true sparsity pattern $\mathcal{S}(A)$ of the Hessian matrix A. By *approximation* we mean that some of the entries (zero/nonzero) of A are mislabelled (i.e., zero labelled nonzero and nonzero labelled zero). The resulting Hessian approximation A_0 by the CPR method will have "incorrect" entries. Suppose the entry $A(i,j) \neq 0$ has been mislabelled as zero in the pattern $\mathcal{S}_0$ and let B_0 be the corresponding compressed Hessian obtained via the CPR method. There are two cases to consider.

1. There is an index $j' \neq j$ such that $\mathcal{S}_0(i,j') \neq 0$ and $\Phi(j) = \Phi(j')$.

2. There does not exist an index $j' \neq j$ such that $\mathcal{S}_0(i,j') \neq 0$ and $\Phi(j) = \Phi(j')$.

From the compressed matrix B_0, we identify missing entries and entries that do not satisfy symmetry and mark both the entries as "flawed". There are two types of flaws:

For case (1), we have $A_0(i,j') \neq A_0(j',i)$ thus violating the symmetry of the Hessian. We term elements at location (i,j) and (j,i) *flawed* (of type 1) if $|A_0(i,j) - A_0(j,i)| > \eta_1$, where η_1 is a positive threshold.

For case (2), $A_0(i,j) = 0$ but $B_0(i,l) \neq 0$ where $\Phi(j) = l$. We term element at location (i,l) of matrix B_0 to be *flawed* (of type 2) if $|B_0(i,l)| > \varepsilon\,\eta_2$, where η_2 is a positive threshold. In the above η_1 and η_2 are user adjustable tolerance to account for errors during floating point operations. As we have already observed, a *'flaw'* is an entry (i,j) such that $A_0(i,j) \neq A_0(j,i)$. Science it is not known which of the two is actually *flawed*, we mark both of them to be *"flawed"*. Moreover a *flawed* entry $A_0(i,j)$ (and $A_0(j,i)$) implies that there are nonzero entries:

- $A_0(i,j) : A_0(i,l) \neq 0$ such that, $\Phi(j) = \Phi(l)$ is not included in the current sparsity pattern of the symmetric matrix.

- $A_0(j,i) : A_0(j,k) \neq 0$ such that $\Phi(i) = \Phi(k)$ is not included in the current sparsity pattern of the symmetric matrix.

The discussion above leads to the following propositions,

Proposition 1. *If $A_0(i,j)$ is a type 1 flaw where $\Phi(j) = l$, then the set possible locations for missing elements is (i,k) where $\Phi(k) = l$ for $k = 1$ to n.*

Proposition 2. *If $B_0(i,l)$ is a type 2 flaw, then the set possible locations for missing elements is (i,k) where $\Phi(k) = l$ for $k = 1$ to n.*

Once all the flaw locations have been identified, we define a *flaw matrix* $\mathcal{F} \in \{0,1\}^{n\times n}$ such that $\mathcal{F}(i,j) = 1$ and $\mathcal{F}(i,j) = 0$ imply, respectively, flaw and no flaw at location (i,j). Let the entries of $\mathcal{F}$ be initialized to 0.

- For each type 1 flaw (i,j), set $\mathcal{F}(i,j') = 1$ where $j' \neq j$ and $\Phi(j) = \Phi(j')$ and $\mathcal{F}(j,i') = 1$ where $i' \neq i$ and $\Phi(i) = \Phi(i')$.
- For each type 2 flaw (i,l), set $\mathcal{F}(i,j) = 1$ where $\Phi(j) = l$.

From our above example, the *col_ind* , *row_ptr* and coloring information Φ of trial pattern $\mathcal{S}_0$ are following.

col_ind

1	3	2	4	1	3	5	2	4	6	3	5	7	4	6	8	5	7	9	6	8	10	7	9	8	10

row_ptr

1	3	5	8	11	14	17	20	23	25

Φ

1	2	3	1	2	3	1	2	3	1

grpind

1	4	7	10	2	5	8	3	6	9

The array Φ indicates the group number or color number for each column. $k = \Phi(j)$ indicates that column j is in group k. Array *grpind* contains the number of columns of each color in partition Φ.

Now we can access the nonzero elements of A through matrix B in the following way. We

Algorithm 2: Accessing nonzeros of matrix A through matrix B

```
1 for i = 1 to n do
2     $index = row\_ptr(i) \dots row\_ptr(i+1) - 1$ ;
3     $j \leftarrow col\_ind(index)$ ;
4     $k \leftarrow \Phi(j)$ ;
5     $a_{i,j} \equiv b_{i,k}/\delta_j$ ;
```

have sparsity pattern for A, we know where $a_{ij} \neq 0$ is mapped to in matrix B [$A(i,j)$ maps to $B(i,\Phi(j))$]. After we have iterated over all rows of the pattern for matrix A, we have identified all flaws of types 1 and any unmarked entry in matrix B_0 must be flaw type 2.

A simple implementation of the marking scheme is to use an array of size n (number of rows). From the sparsity pattern for A, we can compute the number of nonzero entries in each row. For our example the array is,

count

2	2	3	3	3	3	3	3	2	2

Each time a nonzero entry a_{ij} (and a_{ji}) is identified with the corresponding entries in matrix B_0, the count array for the respective rows are decreased by one(1). If the condition of the entry identification for some row index i, $count(i) = -l$, then the i^{th} row of B_0 has l entries are not in matrix A but are nonzero in B_0. Those entries are type 2 *flaws*.

Example:

From our example matrix B_0, we can determine the location of *flaws* in A_0. In the following Figure (3.10) we can see that, out of 10 rows, rows 3, 5 and 9 contain type 1 *flaw* (marked in light-gray) and only row 1 contains type 2 *flaw* (marked in dark-gray).

In type 1 *flaw* column 3 and 9 overlap in row 3 (group 3), 1 and 7 in row 5 (group 1)

$$B_0 \rightarrow \begin{bmatrix} R*\delta_1 & X*\delta_5 & G*\delta_3 \\ R*\delta_4 & B*\delta_2 & . \\ R*\delta_1 & B*\delta_5 & G*\delta_3+Z*\delta_9 \\ R*\delta_4 & B*\delta_2 & G*\delta_6 \\ X*\delta_1+R*\delta_7 & B*\delta_5 & G*\delta_3 \\ R*\delta_4 & B*\delta_8 & G*\delta_6 \\ R*\delta_7 & B*\delta_5 & G*\delta_9 \\ R*\delta_{10} & B*\delta_8 & G*\delta_6 \\ R*\delta_7 & . & G*\delta_9+Z*\delta_3 \\ R*\delta_{10} & B*\delta_8 & . \end{bmatrix}$$

Figure 3.10: Matrix B_0 with *flaws*

and 9 and 3 in row 9 (group 3)of true sparsity pattern.

In type 2 *flaw* the $(1,2)$ element of column 2 does not overlap with any other column in group 2.

Figure (3.11) shows the location of type 1 and type 2 *flaws* in matrix A_0 by using Algorithm 2. Then, the pattern $\mathcal{S}_0$ can be augmented as $\mathcal{S}_1 = \mathcal{S}_0 \bigcup \mathcal{S}\left(\mathcal{F} \bigcap \mathcal{F}^\top\right)$ and the CPR estimation

$$\begin{bmatrix} . & f & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & f & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & f & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & f & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & f & . \\ . & . & . & . & . & . & . & . & . & . \end{bmatrix}$$

Figure 3.11: Location of *flaws* in matrix A_0

of the Hessian with the structurally orthogonal partitioning based on $\mathcal{S}_1$ will yield matrix B_1. In the pattern matrix $\mathcal{S}_1$ some zero entries of the true Hessian matrix may have been mislabelled as nonzero. Such entries will vanish in B_1 and can be easily be identified using the pattern $\mathcal{S}_1$. Removing these spurious entries from $\mathcal{S}_1$ yields the true sparsity pattern of the Hessian. We now formally defined operations on patterns. Let $P \in \{0,1\}^{n\times n}$ and $Q \in \{0,1\}^{n\times n}$ be binary matrices.

Definition 3.3. The intersection $W = P \bigcap Q$

$$W(i,j) = \begin{cases} 1 & \text{if } P(i,j) = Q(i,j) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.4. The union $V = P \bigcup Q$

$$V(i,j) = \begin{cases} 0 & \text{if } P(i,j) = Q(i,j) = 0 \\ 1 & \text{otherwise} \end{cases}$$

Thus we obtain the following algorithm to calculate *flaws* locations where, matrix A_0

Algorithm 3: Compute *flaw* locations

```
Input  : The matrix $\mathcal{S}_0$
         Array $\Phi$ that gives the column partitioning of symmetric trial pattern $\mathcal{S}_0$
         The matrix $B_0$
Output: Set of flaws locations
1  for each $A_0(i,j) \neq A_0(j,i)$ with $J \geq i$ do              // computing type 1 flaw
2      if $|A_0(i,j) - A_0(j,i)| > \eta_1$ then
3          if $i$=$j$ then
4              add $(i,i)$ to the set of flaw type 1
5          if $i \neq j$ then
6              add $(i,j)$ and $(j,i)$ to the set of type 1 flaw
7  for each $B_0(i,k) \neq 0$ do                                     // computing type 2 flaw
8      for each $A_0(i,j) = 0$ do
9          if $\Phi(j) = k$ and $|B_0(i,k)| > \varepsilon\,\eta_2$ then
10             add $(i,j)$ to the set of type 2 flaw
```

referred in Algorithm 3 is implicitly obtained via Algorithm 2.

3.3.2 Asymptotic Analysis

In Algorithm 3, lines $(1-6)$ require $O(np)$ time to calculate type 1 *flaw*. The time for calculating type 2 *flaw* (lines $(7-10)$) is also $O(np)$. The resulting computational complexity of Algorithm 3 for calculating *flaw* locations is $O(np)$.

3.4 Data Structure for Flaws

Our implementation relies heavily on the use of efficient sparse data structures and operations on sparse matrices. The implementation of union and intersection operations as well as the *flaw* calculations use sparse data structures based on compressed sparse row (CSR) and compressed sparse column (CSC) representations which allow a cache-friendly performance to minimize cache misses. The flaw matrix need not be stored explicitly; it can be generated as needed. It is also possible to have one or more *flaws* in row i. We have a data structure similar to our row-oriented sparsity pattern data structure (CRS). To explain the data structure we consider the previous example where, n=10 and $|\Phi| = 3$. The

Table 3.1: Data structure for type 1 *flaw*

row	*flaw_groups*
3	3
5	1
7	2
9	3

g_index				$flaw\_ptr$				$flaw\_row$			
3	1	2	3	1	2	3	4	3	5	7	9

Table 3.2: Data structure for type 2 *flaw*

row	*flaw_groups*
1	2

g_index	$flaw\_ptr$		$flaw\_row$
2	1	2	1

column labeled *flaw_groups* in Table (3.1) and Table (3.2) denote the group number ($\Phi(j)$) of the *"flawed"* entry in each row i where j is the column index as described in Algorithm 2. The array $flaw\_row$ contains indices of rows that has at least one *flaw*. Here the row indices are stored in sorted order (increasing). The array $flaw\_ptr$ contains indices of array g_index such that, $g\_index(k)$, $k = flaw\_ptr(i) \ldots flaw\_ptr(i+1) - 1$, gives the group or color index of each *flaw* in row $flaw\_row(i)$. The data structure for all *flaws* $\mathcal{F}$ in *CRS* format as follows,

$fcol\_index$

2	5	8	3	9	1	4	10	2	8	3	6

$frow\_ptr$

1	4	6	9	11

Figure 3.12: Data structure of *flaws* in *CRS* format

In Figure (3.12), $frow\_ptr$ represents the starting index of each row and $fcol\_ind$ represents the column indices of each row.

The *flaws* sparsity pattern $\mathcal{S}\left(\mathcal{F} \bigcap \mathcal{F}^\top\right)$ in matrix form is presented in the following Figure (3.13) where the $\mathcal{F} \bigcap \mathcal{F}^\top$ operation eliminates all the nonsymmetry entries from the flaw matrix $\mathcal{F}$.

$$\mathcal{F} \rightarrow \begin{bmatrix} \cdot & f & \cdot & \cdot & f & \cdot & \cdot & f & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ f & \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & f \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & f & \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix} \xrightarrow{\mathcal{F} \bigcap \mathcal{F}^\top} \begin{bmatrix} \cdot & \cdot & \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & f & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ f & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & f & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}$$

Figure 3.13: *flaw* matrix $\mathcal{F}$

Finally, the pattern $\mathcal{S}_0$ can be augmented as $\mathcal{S}_1 = \mathcal{S}_0 \bigcup \mathcal{S}\left(\mathcal{F} \bigcap \mathcal{F}^\top\right)$ to updated the sparsity pattern.

$$\begin{bmatrix}
\mathbf{R} & . & \mathbf{G} & . & f & . & . & . & . & . \\
. & \mathbf{B} & . & \mathbf{R} & . & . & . & . & . & . \\
\mathbf{R} & . & \mathbf{G} & . & \mathbf{B} & . & . & . & f & . \\
. & \mathbf{B} & . & \mathbf{R} & . & \mathbf{G} & . & . & . & . \\
f & . & \mathbf{G} & . & \mathbf{B} & . & \mathbf{R} & . & . & . \\
. & . & . & \mathbf{R} & . & \mathbf{G} & . & \mathbf{B} & . & . \\
. & . & . & . & \mathbf{B} & . & \mathbf{R} & . & \mathbf{G} & . \\
. & . & . & . & . & \mathbf{G} & . & \mathbf{B} & . & \mathbf{R} \\
. & . & f & . & . & . & \mathbf{R} & . & \mathbf{G} & . \\
. & . & . & . & . & . & . & \mathbf{B} & . & \mathbf{R}
\end{bmatrix}$$

Figure 3.14: Updated sparsity pattern S_1

Figure (3.14) shows the updated sparsity pattern $\mathcal{S}_1$ where all the missing elements from matrix A are located. In the absence of a "good" start pattern $\mathcal{S}_0$, we use a multilevel heuristic with randomly generated sparsity patterns and iterate until the true pattern is detected.

Chapter 4

Detection of Hessian Matrix Sparsity Pattern

In this chapter, we first briefly describe a greedy partitioning algorithm which finds a structurally orthogonal partitioning of the columns of an input sparse matrix. The columns are preprocessed with the ordering algorithm Smallest-Last Ordering (SLO). Empirical evidence [25, 31, 27, 28] suggest that the greedy algorithm yields better partitions (with fewer groups) when the columns are processed in certain order. The SLO appears to give good results [10, 11, 13, 17]. In the second part of the chapter we adopt a multilevel algorithm from [7] to detect the true sparsity pattern of Hessian matrices.

4.1 DSJM toolkit

DSJM is a software toolkit for direct determination of sparse Jacobian matrices and can also be used to find consistent partitions of large sparse matrices [24]. The toolkit is written in C++. DSJM uses efficient sparse data structures (variants of Compressed Row and Compressed Column) to implement the ordering and coloring algorithms which allow the kernel operations to perform in a cache-friendly way to minimize cache misses due to irregular data access. DSJM applies greedy coloring to the vertices in some chosen order which is more efficient. The ordering methods used in DSJM are: Largest-First Ordering (LFO), Smallest-Last Ordering (SLO), Incidence-Degree Ordering (IDO), Saturation-Degree Ordering (SDO), Recursive-Largest-First (RLF) and a hybrid approach MRLF-SLO based on RLF and SLO. In our implementation we use the Smallest-Last Greedy Coloring (SLO) for partitioning the columns into structurally orthogonal groups.

4.1.1 Greedy coloring method

Curtis, Powell and Reid proposed, a greedy partitioning heuristic to find a structurally orthogonal mapping Φ [14]. Let $G(H) = (V,E)$ be the graph representing matrix H where,

- $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices corresponding to columns and $e = \{v_i, v_j\} \in E$ if $H(:,i)$ and $H(:,j)$ are structurally dependent, i.e. there is an index k such that $H(k,i) \neq 0$ and $H(k,j) \neq 0$.

Let $N(v)$ be the set of neighbors of vertex $v \in V$ in $G = (V,E)$, i.e. $N(v) = \{u \in V | u \neq v, \{u,v\} \in E\}$. The degree of v in G is denoted by $deg(v) = |N(v)|$. A greedy sequential coloring algorithm as follows.

```
Algorithm: Sequential Coloring
Input  : order array containing a permutation of columns 1...n
Output: color array containing the color of columns 1...n
1 for i = 1 to n do
2     $color(i) = n$
3 for j = 1 to n do
4     $col \leftarrow order(j)$ ;
5     Find $N(col)$ and mark their colors ;
6     Let c be the smallest color index that has not been assigned to any column in
        $N(col)$, assign color index c to column col ;
```

The major computational cost of the above algorithm is $O(\sum_{\{i|a_{ij}\neq 0\}} \rho_i)$ to find $N(col)$. For all columns in *order* array the cost is proportional to $O(\sum_{i=1}^{n} \sum_{\{i|a_{ij}\neq 0\}} \rho_i) = O(\sum_{i=1}^{n} \rho_i^2)$. The total time complexity of sequential coloring algorithm is $O(\sum_{i=1}^{m} \rho_i^2)$ [24].
The following Figure (4.1) shows the structurally orthogonal coloring graph $G(H)$ of matrix H where columns $(v_1, \ldots, v_6)$ are grouped into three colors (1, 2 and 3).

$$H = \begin{bmatrix} h_{11} & . & h_{13} & . & . & . \\ . & . & . & . & h_{25} & . \\ h_{31} & . & h_{33} & . & . & h_{36} \\ . & . & . & h_{44} & . & h_{46} \\ . & h_{52} & . & . & . & h_{56} \\ . & . & h_{63} & h_{64} & h_{65} & . \end{bmatrix}$$

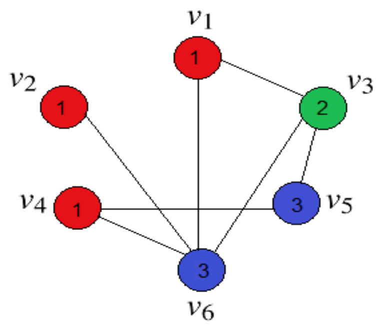

Figure 4.1: Graph $G(H)$ representing the column partition of matrix H.

4.1.2 Smallest-Last Ordering

Let the vertices $V' = \{v_n, v_{n-1}, ..., v_{i+1}\}$ be already ordered. The i^{th} vertex in SLO is an unordered vertex u such that $deg(u)$ is minimum in $G[V \setminus V']$ where, $G[V \setminus V']$ is the graph obtained from G by deleting the vertices of set V' from V. Following algorithm shows the major computational steps of Smallest-Last ordering.

```
Algorithm: SLO
   Input  : degree array containing the degree information for columns 1...n
   Output: order array containing the ordered columns 1...n
1  Construct a priority queue from degree information ;
2  while the priority queue is not empty do
3      Select column $j$ from the priority queue with the minimal degree in $G[V \setminus V']$ ;
4      Place $j$ in the order array ;
5      Remove column $j$ from priority queue ;
6      Compute $N(j)$ for column $j$ ;
7      Update the degree for the neighbors of column $j$;
```

In SLO the major computational steps are:

1. Find a column j with the minimal degree in $G[V \setminus V']$ at a cost of $O(\log j)$.

2. $N(j)$ computation at a cost of $O(\sum_{\{i|a_{ij}\neq 0\}} \rho_i)$.

Over all columns step 1 becomes $O(\sum_{i=1}^{n} \log j)$ which is $O(n \log n)$ and step 2 becomes $O(\sum_{i=1}^{n} \sum_{\{i|a_{ij}\neq 0\}} \rho_i) = O(\sum_{i=1}^{n} \rho_i^2)$. Updating are done along with the same loop, so the computational complexity of SLO algorithm is also $O(\sum_{i=1}^{n} \rho_i^2)$ [24].

4.2 The Multilevel Algorithm

In our research, we investigate a multilevel algorithm for automatic detection of sparsity pattern of sparse Hessian matrices. The following problems will be considered:

1. Given a starting guess, investigate algorithms to determine the sparsity pattern while minimizing the number of matrix-vector products (gradient evaluation with FD calculation).

2. Given certain known numerical/structural properties on the Hessian entries investigate algorithms to determine the sparsity pattern while minimizing the number of matrix-vector products.

3. Implement the algorithms on serial computers.

In chapter 3, we observed that with a “good guess” on the true sparsity pattern the true pattern can be detected using two CPR applications: one corresponding to the guess $\mathcal{S}_0$ and one corresponding to the augmented pattern $\mathcal{S}_1$. By a “good guess” we mean that only a very small fraction of nonzero elements are missing from the guess. We also have assumed that the diagonal elements of the Hessian can be computed by alternative means. In this chapter we address these drawbacks. The methods we describe here draws heavily from [7]. In absence of a ‘guess’ one can use a random sparsity pattern for $\mathcal{S}_0$ and apply the method of chapter 3. One problem with this approach is that the augmented pattern $\mathcal{S}_1$ may become so dense that all advantage of using the CPR is lost. A key observation is that the partitioning algorithm is very efficient when applied to sparse patterns. Also, the main computational operation is the evaluation of the gradient. A suitable strategy to minimize the number of gradient evaluations could be:

a) Use a small number of random sparsity patterns and use the CPR method to determine the set of *flaws* for each pattern. We then take intersection of all such *flaws* to yield the location of missing elements. Then augment the guess $\mathcal{S}_0$ with the missing elements as computed. To avoid dense patterns one can terminate the *flaw* computation when the number of possible *flaw* locations determined so far exceeds a predetermined threshold. The algorithm can now restart with a different set of randomly generated sparsity pattern. The number of times this algorithm is repeated before a reasonably sparse augmenting pattern has been determined must be very small. We call each application of the algorithm as "level".

   To summarize, each level corresponds to a small set of randomly generated sparsity pattern. The intersection of the corresponding *flaws* gives the augmenting pattern. The algorithm is terminated for the current level if the augmenting pattern is unacceptably dense.

b) When the diagonal of the Hessian matrix can not be computed directly, one can use simple "voting" strategy. To clarify, diagonal entries at each level are noted and the determination of whether the entry is a zero or nonzero is settled by a simple majority assumed over all the levels.

In our implementation we adapt the multilevel heuristic of [7] whereby the initial pattern is augmented with a small number (say k) of random patterns and using the CPR method a *flaw* matrix is defined as the intersection of the *flaw* matrices obtained for each of k patterns. It is expected that the flaw matrix thus obtained will be sufficiently sparse such that the pattern $\mathcal{S}_1$ is a good approximation of the true pattern and a superset of the true sparsity pattern. Finally, with one more CPR calculation the spurious flaws elements are removed to reveal the true pattern.

4.2.1 Voting between Levels

In the multilevel algorithm, we can find the flaw locations by comparing individual elements of trial matrix $\mathcal{S}_k$ from level to level and using a voting scheme to decide which diagonal entries are probably flawed [7].

At each of the k levels of multilevel algorithm, we record the value of $B(i,l)$ where $\Phi(i) = l$ for $i = 1,2,\ldots,n$. Whether the diagonal element at location (i,i) is a zero or not is decided by the value (with a given tolerance) taken by the majority of elements across k levels. If the majority of the elements $\mathcal{S}_l(i,i)$, $l = 0,1,\ldots.,k$ take the same value within a threshold, we can assume with some certainty that the value is the correct value for $H(i,i)$. The voting scheme becomes better as more levels are utilized. The algorithm to calculate *flaws* locations using voting between levels is given below.

Algorithm 4: Compute *flaw* locations using voting between levels

```
Input  : The trial pattern $\mathcal{S}_k$
         Array $\Phi_k$ that gives the partitioning of symmetric trial pattern $\mathcal{S}_k$
         The compressed matrix $B_k$
Output: Set of all flaws locations
1  for $i = 1,\ldots.n$ do                                  // computing type 1 flaw
2      if $|A_l(i,i) - A_{l-1}(i,i)| < \eta_3(|A_l(i,i)| + |A_{l-1}(i,i)|)$ for majority of the elements
        where, l=0,1,....,k then
3          for $l = 0,\ldots,k$ do
4              if $|A_l(i,i) - A_l(i,i)| > \eta_1$ then
5                  add $(i,i)$ to the set of flaw type 1
6      else
7          for $l = 0,\ldots,k$ do
8              add $(i,i)$ to the set of flaw type 1
9  for each $A_k(i,j) \neq A_k(j,i)$ with $j \geq i$ at level $k$ do
10     if $i \neq j$ and $|A_k(i,j) - A_k(j,i)| > \eta_1$ then
11         add $(i,j)$ and $(j,i)$ to the set of type 1 flaw
12 for each $B_k(i,k) \neq 0$ at level $k$ do               // computing type 2 flaw
13     for each $A_k(i,j) = 0$ do
14         if $\Phi(j) = k$ and $|B_k(i,k)| > \varepsilon\,\eta_2$ then
15             add $(i,j)$ to the set of flaw type 2
```

Two diagonal elements $A_l(i,i)$ and $A_{l-1}(i,i)$ are considered equal if, $|A_l(i,i) - A_{l-1}(i,i)| < \eta_3(|A_l(i,i)| + |A_{l-1}(i,i)|)$ for some positive threshold η_3. In our implementation, we never expand the Hessian approximation A_k into full matrix rather we access A_k by using matrix B_k and the sparsity structure of trial pattern $\mathcal{S}_k$.

Here the type-2 *flaws* do not depend on the voting technique, at each level we recompute type-1 *flaws* at all previous levels using the currently available information. The selection of nonzero constants and threshold values will affect algorithm efficiency. For example a problem can arise if the nonzeros have the same values. If $\delta_i = \delta_j$ for all i,j, the computed location for any flawed element in H will be same for the equal nonzero values. Thus in our implementation,

- $\delta_i \neq \delta_j$ for all i,j and $\delta_i = r_i \times 10^{-12}$, where r_i is a randomly selected number between 0.5 to 2.
- $\eta_i = 10^{-12}$, for $i = 1,2$ and 3.

4.2.2 Asymptotic Analysis

In Algorithm 4, lines $1-8$ executes the voting scheme n times between levels 0 to k for calculating type 1 *flaws*. The time complexity of voting between level (line 2) is $O(k)$. The time for adding (i,i) *flaws* between level (line 3-8) is also $O(k)$. The total computational complexity of voting scheme is $O(nk^2)$.

Line (9-11) require $O(np)$ time to calculate type 1 *flaw* for $i \neq j$. There is no voting for type 2 *flaws*, so the time complexity of calculating type 2 *flaw* (line $12-15$) is still $O(np)$ as in Algorithm 3. The resulting time complexity for Algorithm 4 is $O(nk^2) + O(np)$.

Finding all Possible *flaws*

If the randomly generated sparsity pattern has enough randomizing effect then the locations of spurious elements will be significantly different and the number of spurious elements in $(\mathcal{F}_0 \bigcap \mathcal{F}_0^\top) \bigcap (\mathcal{F}_1 \bigcap \mathcal{F}_1^\top)$ will be considerably reduced from the number present in

$(\mathcal{F}_0 \bigcap \mathcal{F}_0^{\top})$.

The *flaw* pattern at level k is,

$$\mathcal{S}(\mathcal{F}) = \mathcal{S}\left(\mathcal{F}_0 \bigcap \mathcal{F}_0^{\top} \bigcap \mathcal{F}_1 \bigcap \mathcal{F}_1^{\top} \bigcap \ldots \bigcap \mathcal{F}_k \bigcap \mathcal{F}_k^{\top}\right)$$

It is not necessary to calculate or store the intermediate patterns $\mathcal{F}_i$ for $i = 1, \ldots, k-1$ and $\mathcal{S}(\mathcal{F})$ because $\mathcal{S}(\mathcal{F})$ can be directly computed from the *flaws*. Algorithm 5 describes

Algorithm 5: Compute the set of all possible *flaws* $\mathcal{S}(\mathcal{F})$

```
Input  : Array $\Phi_k$ that gives the column partitioning of symmetric pattern $\mathcal{S}_k$
         flaw locations $\mathcal{F}_k$
Output: The set of all possible flaws $\mathcal{S}(\mathcal{F})$
1 Set $\mathcal{S}(\mathcal{F}) = \phi$ ;
2 for each flaw $(i, j)$ do
3     for each element $(i, l)$ where $l \in \Phi_k(j)$ and $k > i$ do
4         Add $(i, l)$ and $(l, i)$ to the set $\mathcal{S}(\mathcal{F})$ ;
```

the pattern of flaw matrix $\mathcal{F}$ according to *Proposition* 1 and *Proposition* 2 and based on the *flaws* computed by Algorithm 4. We emphasize that the *flaw* matrix need not be constructed explicitly and is shown here for the purpose of illustration.

Now we can present the complete multilevel algorithm for the determination of Hessian matrix.

4.2.3 Multilevel Algorithm to Determine the Sparsity Structure of Hessian Matrix H

Algorithm: Multilevel algorithm to determine the sparsity structure of Hessian matrix

```
    Input  :  A symmetric trial pattern $\mathcal{S}_0$
              An estimation on the number of missing elements $n_{estimate}$
    Output: The final true pattern $\mathcal{S}(H)$
 1  if $I \not\subseteq \mathcal{S}_0$ then
 2      $\mathcal{S}_0 = \mathcal{S}_0 \cup I$ ;
 3  $k = 0$ ;
 4  if $nnz(\mathcal{S}_0) < k_1 n_{estimate}$ then
 5      $\mathcal{S}_0 = \mathcal{S}_0 + \mathcal{R}_0$
 6  Use DSJM toolkit to partition the columns of $\mathcal{S}_0$ into $\Phi_0$ ;
 7  Compute matrix $B_0$ using the CPR algorithm with $\mathcal{S}_0$, $S$ and $\Phi_0$; // using
     Algorithm 1;
 8  Compute the *flaw* locations $\mathcal{F}_0$ and set $n_{estimate} = nnz(\mathcal{F}_0)$; // using Algorithm 4;
 9  Compute the set of *flaws* $\mathcal{S}(\mathcal{F}) = \mathcal{S}\left(\mathcal{F}_0 \bigcap \mathcal{F}_0^\top\right)$ ;
10  while $nnz(\mathcal{S}(F))/\alpha > n_{estimate}$ do
11      $k = k + 1$ ;
12      Generate a random symmetric pattern $\mathcal{R}_k$ where, $nnz(\mathcal{R}_k) \geq k_2 n$ ;
13      $\mathcal{S}_k = \mathcal{S}_{k-1} \cup \mathcal{R}_k$ where, $nnz(\mathcal{R}_k) + nnz(\mathcal{S}_{k-1}) \geq k_1 n_{estimate}$;
14      Use DSJM toolkit to partition the columns of $\mathcal{S}_k$ into $\Phi_k$;
15      Compute matrix $B_k$ using the CPR algorithm with $\mathcal{S}_k$, $S$ and $\Phi_k$; // using
         Algorithm 1 ;
16      Compute the *flaw* locations $\mathcal{F}_k$; // using Algorithm 4;
17      Estimate the number of missing elements, $n_{estimate} = nnz(\mathcal{F}_k)$ ;
18      if $k < 2$ then
19          go to step 9
20      Compute the set of *flaws* $\mathcal{S}(\mathcal{F}) = \mathcal{S}\left(\mathcal{F}_0 \bigcap \mathcal{F}_0^\top \bigcap \mathcal{F}_1 \bigcap \mathcal{F}_1^\top \bigcap \ldots \bigcap \mathcal{F}_k \bigcap \mathcal{F}_k^\top\right)$ ;
21  $k = 0$ ;
22  $\mathcal{S}_0 = \mathcal{S}_0 \bigcap \mathcal{S}(\mathcal{F})$ ;
23  Use DSJM toolkit to partition the columns of $\mathcal{S}_0$ into $\Phi_0$ ;
24  Compute matrix $B_0$ using the CPR algorithm with $\mathcal{S}_0$, $S$ and $\Phi_0$; // using
     Algorithm 1;
25  Compute the flaw locations $\mathcal{F}_0$; // using Algorithm 4;
26  Estimate the number of missing elements, $n_{estimate} = nnz(\mathcal{F}_0)$ ;
27  Get the final pattern $\mathcal{S}(H)$ by eliminating spurious entries from $\mathcal{S}_0$; // using
     Algorithm 6;
28  if $n_{estimate} = 0$ then
29      exit;
30  else
31      go to step 11 ;
```

Here we start with an initial estimation on the number of missing elements $n_{estimate}$ based on input trial pattern $\mathcal{S}_0$, which is used to determine the the number of nonzeros for the starting guess pattern. Later we update $n_{estimate}$ by the number of nonzero elements in *flaw* matrix $\mathcal{F}$. We use a random number generator to produce pairs of integers $(i,j)_k$, $k = 1,2,....m/2$ (with each integer between 1 to n) to generate a symmetric pattern R_k with m entries. The reason of taking random pattern R_k is to make different guess pattern by $\mathcal{S}_k = \mathcal{S}_{k-1} \cup \mathcal{R}_k$ at each level of computation. We should consider some adjustment before selecting these values as proper selection of the values of these parameters can lead to improved efficiency.

- The number of nonzero elements in $\mathcal{S}_k$ should be at least a small multiple of $n_{estimate}$, i.e. $nnz(\mathcal{S}_k) \geq k_1 n_{estimate}$
- The number of nonzero elements in $\mathcal{R}_k$ should be at least a small multiple of n, i.e. $nnz(\mathcal{R}_k) \geq k_2 n$ to randomize the pattern of *flaws*.
- The *while* loop should execute until the number of nonzero elements in *flaw* matrix pattern is significantly small than $n_{estimate}$, i.e. $nnz(\mathcal{S}(F))/\alpha > n_{estimate}$.

Where $k_1 \geq 2$, $k_2 \geq 1$ and $\alpha \geq 5$. Line $(4-5)$ execute when no initial guess pattern supplies or the number of nonzero elements in $\mathcal{S}_0$ is less then a small multiple of $n_{estimate}$, i.e. $nnz(\mathcal{S}_0) < k_1 n_{estimate}$.

To Compute the set of *flaws* $\mathcal{S}(\mathcal{F})$ we do not need to calculate or store the intermediate patterns $\mathcal{F}_k$ because $\mathcal{S}(\mathcal{F})$ can be directly calculated from the *flaws*.

In the pattern matrix $\mathcal{S}_0$ (at line 22 of multilevel algorithm) some zero entries of the true Hessian matrix may have been mislabelled as nonzero. Such entries will vanish in B_0 (at line 24) and can be easily be identified using the pattern $\mathcal{S}_0$. Removing these spurious entries from $\mathcal{S}_0$ yields the true sparsity pattern of the Hessian. Algorithm to eliminate spurious zeros from B_0 as follows,

Algorithm 6: Eliminating spurious entries from $\mathcal{S}_0$

```
Input  : Matrix $\mathcal{S}_0$
         Array $\Phi$ that gives the column partitioning of symmetric pattern $\mathcal{S}_0$
         Matrix $B_0$
Output: The final pattern $\mathcal{S}(H)$
1 for each $\mathcal{S}_0(i,j) \neq 0$ where $j \geq i$ do
2     if max $(|A_0(i,j)|, |A_0(j,i)|) < \eta_0$ then
3         $\mathcal{S}_0(i,j) = 0$ ;
```

where η_0 is a positive threshold used to find the spurious zeros in A_0 ($\eta_0 = 10^{-6}$ in our implementation) and we access A_0 by using matrix B_0.

4.2.4 Asymptotic Analysis

In the multilevel algorithm, most computationally expensive operations are performed within the *while* loop (lines 10-20). The DSJM toolkit requires $O(\sum_{i=1}^{m} \rho_i^2)$ times to partition the columns of $\mathcal{S}_k$ (line 14) and the Algorithm 1 needs $O(nnz)$ times (line 15) to compute matrix B_k. The time complexity for Algorithm 4 is $O(nk^2) + O(np)$ to compute the *flaw* locations (line 16). The time for Eliminating spurious entries from $\mathcal{S}_0$ in line 27 is $O(nnz)$ where *nnz* is the number of nonzeros in $\mathcal{S}_0$ (in Algorithm 6).

Chapter 5

Numerical Experiments

In this chapter, we present numerical results of applying algorithms proposed in this thesis on practical instances. In Section 5.1 we give details of test data sets that we have used in our experiments. We describes the numerical experiments with the iterative algorithm in Section 5.2. In Section 5.3, we provide our findings.

5.1 Test Data Sets

In this section, we discuss about our data sets. The data set in Table 5.1 is for the matrices which are collected from Matrix Market Collection [3]. The data set in Table 5.2 is obtained from University of Florida Matrix Collection [5]. In tables, Table 5.1 and Table 5.2 we provide the name of the matrix under the column labeled Matrix, the columns labeled *n* is used to represent the number of columns and rows (square matrix) and *nnz* is used to represent the total number of nonzeroes in the matrix.

Table 5.1: Matrix Data Sets-1

Matrix	*n*	*nnz*
bcspwr05	443	1,623
dwt_592	592	5,104
nos6	675	3,255
young1c	841	4,089

Continued on next page

Table 5.1 – continued from previous page

Matrix	*n*	*nnz*
sherman1	1,000	3,750
rdb1250	1,250	7,300
lshp1561	1,561	10,681
fidap004	1,601	32,287
plat1919	1,919	32,399
rdb2048	2,048	12,032
orsreg_1	2,205	14,133
zenios	2,873	27,191
pde2961	2,961	14,585
bcsstk21	3,600	26,600
e20r0000	4,241	131,556
fidapm09	4,683	95,053
mhd4800b	4,800	27,520
sherman3	5,005	20,033
bcspwr10	5,300	21,842
fidap018	5,773	69,335
fidap015	6,867	96,421
e30r1000	9,661	306,356
bcsstk17	10,974	428,650
bcsstk18	11,948	149,090
fidapm29	13,668	186,294

Continued on next page

Table 5.1 – continued from previous page

Matrix	*n*	*nnz*
bcsstk29	13,992	619,488
bcsstk25	15,439	252,241
e40r5000	17,281	553,956
bcsstk30	28,924	2,043,492

Table 5.2: Matrix Data Sets-2

Matrix	*n*	*nnz*
662_bus	664	2,274
dwt_758	758	5,994
rdb800l	800	4,640
olm1000	1,000	4,994
Jagmesh3	1,089	4,661
Jagmesh5	1,180	4,951
Jagmesh8	1,141	7,465
dwt1242	1,242	7,213
lshp1270	1,270	5,533
Jagmesh9	1,349	5,654
Jagmesh4	1,440	6,052
bscpwr06	1,454	5,300
bscpwr08	1,624	6,050
filter2D	1,668	10,750

Continued on next page

Table 5.2 – continued from previous page

Matrix	*n*	*nnz*
ex33	1,733	22,189
watt_1	1,856	11,488
G26	2,000	39,980
t2dal_*a*	4,257	37,465
nasa4704	4,704	104,756
EX5	6,545	196,360
Kuu	7,102	340,200
G65	8,000	32,000
delaunay_*n*13	8,192	49,094
aft01	8,205	125,567
nemeth02	9,506	394,808
wing_*nodal*	10,937	150,976
linverse	11,999	95,977
stokes64s	12,546	140,034
barth5	15,606	107,362
gyro_*m*	17,361	340,431
Trefethen_20000*b*	19,999	554,435
t3dl_*e*	20,360	20,360
tube1	21,498	897,056

5.2 Test Environment

Numerical Experiments with data sets in Table (5.1) and Table (5.2) were done on an AMD PC with AMD Opteron(tm) Processor 4284, 16 GB RAM and 2048kB L2 cache running 64 bit Linux.

5.3 Test Results

Our implementation relies heavily on the use of efficient sparse data structures and operations on sparse matrices. We use the package DSJM [24] to compute the structurally orthogonal partitioning and use difference approximation (Equation (1.1)) to compute matrix B corresponding to a given sparsity pattern. Matrix B is stored as a dense matrix since most of its entries are nonzero. The union and intersection operations as well as the computation of flaw matrix $\mathcal{F}$ use sparse data structures based on compressed sparse row (CSR) and compressed sparse column (CSC) representations [24]. Our test results are shown in Table 5.3 and 5.4 with Matrix Market collection and University of Florida Sparse Matrix Collection respectively. In both tables, $\rho_{\max}$ represents maximum number of nonzeroes in any row of the matrices. For our test matrices the value of $\rho_{\max}$ is between $5-218$. Entries in column labelled `ncolor` denote the number of column groups in the structurally orthogonal partition of the true Hessian. The total number of gradient evaluations to detect the sparsity pattern for each test problem is listed under column `nfeval`.

5.3.1 Numerical Experiments

Table 5.3: Computational Cost for Sparsity Detection for data set 1

Matrix	n	ρ_{max}	`ncolor`	`nfeval`	`nCPR`
bcspwr05	443	10	10	154	8
nos6	675	5	5	151	8

Continued on next page

Table 5.3 – continued from previous page

Matrix	n	ρ_{max}	`ncolor`	`nfeval`	`nCPR`
young1c	841	5	5	162	10
rdb1250	1,250	6	8	297	9
bcsstm12	1,473	22	26	713	11
lshp1561	1,561	7	8	368	10
plat1919	1,919	19	24	780	11
rdb2048	2,048	6	9	273	9
orsreg_1	2,205	7	11	239	9
zenios	2,873	14	48	1,095	11
bcsstk21	3,600	9	14	535	11
e20r0000	4,241	62	69	2,516	14
fidapm09	4,683	37	45	1,567	12
mhd4800b	4,800	10	10	497	11
bcspwr10	5,300	14	14	329	12
s1rmt3m1	5,489	48	50	2,864	13
fidap018	5,773	18	22	1,277	14
fidap015	6,867	18	22	1,297	14
e30r1000	9,661	62	70	2,998	12
bcsstk17	10,974	150	150	3,580	11
bcsstk18	11,948	49	49	1,680	13
bcsstk29	13,992	71	72	5,069	11
bcsstk25	15,439	59	59	2,950	15

Continued on next page

Table 5.3 – continued from previous page

Matrix	*n*	ρ_{max}	`ncolor`	`nfeval`	`nCPR`
e40r5000	17,281	62	70	3,383	12
bcsstk30	28,924	218	219	9,131	13

Table 5.4: Computational Cost for Sparsity Detection for data set 2

Matrix	*n*	ρ_{max}	`ncolor`	`nfeval`	`nCPR`
662_bus	664	10	20	214	6
dwt_758	758	11	12	342	7
rdb800l	800	6	10	286	7
olm1000	1,000	6	11	241	7
Jagmesh3	1,089	7	16	284	8
Jagmesh5	1,180	7	8	246	7
Jagmesh8	1,141	7	10	324	8
dwt1242	1,242	12	12	409	10
lshp1270	1,270	7	8	256	7
Jagmesh9	1,349	7	8	251	8
Jagmesh4	1,440	7	8	263	8
bscpwr06	1,454	13	15	293	7
bscpwr08	1,624	14	14	451	11
filter2D	1,668	9	11	589	11
ex33	1,733	18	22	825	13
watt_1	1,856	7	17	259	6

Continued on next page

Table 5.4 – continued from previous page

Matrix	n	ρ_{max}	`ncolor`	`nfeval`	`nCPR`
G26	2,000	40	82	1,504	9
t2dal_*a*	4,257	9	12	894	12
nasa4704	4,704	42	47	1,805	15
EX5	6,545	120	138	3,205	13
Kuu	7,102	96	108	4,744	11
G65	8,000	19	19	318	12
delaunay_*n*13	8,192	12	14	578	11
aft01	8,205	21	25	1,367	11
nemeth02	9,506	52	56	3,739	13
wing_*nodal*	10,937	28	37	1,520	9
linverse	11,999	9	11	764	15
stokes64s	12,546	11	20	1,083	13
barth5	15,606	11	11	1,235	14
gyro_*m*	17,361	120	120	4,287	14
Trefethen_20000*b*	19,999	84	84	4,110	16
t3dl_*e*	20,360	1	3	3	2
tube1	21,498	48	67	5,251	11

5.3.2 Summary of Experimental Results

For the above numerical experiments the inputs of our multilevel algorithm are,

- a band matrix of size $n \times n$ with bandwidth 3 as the input symmetric trial pattern $\mathcal{S}_0$ and
- $n_{estimate} = nnz(H) - 3 * n$ where $nnz(H)$ is the number of nonzero entries in the true

pattern $H \in \mathbb{R}^{n\times n}$.

Line 5 executes to enlarge the input trial pattern $\mathcal{S}_0$ if $nnz(\mathcal{S}_0) < k_1 n_{estimate}$ where $\mathcal{R}_0$ is a band matrix with bandwidth 5. We have done our experiments by setting $k_1 = 2$, $k_2 = 1$ and $\alpha = 5$ in multilevel algorithm. If we increase the value of k_2 to 3 it decreases the number of level but increase the number of gradient evaluations. The same behavior is observed if we increase the value of α.

Number of Iterations

Number of iterations the *while* loop (lines $10-18$) goes through to obtain the true sparsity pattern is between $1-5$. For the first time the while loop iterates 3 or 4 times for most of the test matrices. For some cases the number is 5, e.g., e20r000, ex5 (when the test matrix is less sparse). When the number of nonzeros are very small, the first iteration of *while* loop is 2 or 1. The number of while loop iterations decreases to 1 as the true sparsity pattern reveals with the outer iteration (line 29). The number of outer iterations for multilevel algorithm is between $1-4$. The outer iteration is 1 or 2 only for small number of nonzeros (approximately $nnz < 30,000$). Those iteration numbers depend on the randomly generated sparsity pattern $\mathcal{R}_k$ at each level. If the randomly generated sparsity pattern is a good guess of the true sparsity pattern the number of iteration will be small.

Number of CPR calls

From the number of iterations we can also find the number of calls to the CPR algorithm. For our test results shown above the number of CPR calculation (`nCPR`) is between $7-16$. For small number of nonzeros there are $7-13$ CPR calls and for large number of nonzeros entries there are $11-16$ CPR calls in total.

Number of Gradient Evaluations

The number of gradient evaluation `nfeval` increases inside the *while* loop iteration and decreases for the outer loop iteration.

From the above tables (Table 5.3 and Table 5.4), we can make the following statements about our experimental results

- It is clear that the number of calls to the CPR (`nCPR`) to detect the sparsity pattern is quite modest (relative to n) and is independent of the matrix dimension.
- Furthermore, for all the test matrices, the true pattern of the Hessian (with voting heuristic for the diagonal elements) is obtained within $2-4$ outer iterations.
- We note that $\rho_{\max}$ is a lower bound on the number of groups in a structurally orthogonal partition.
- On average, the number of function evaluations required in the proposed method is approximately one-fourth of the number of columns n for large matrices.

Chapter 6

Conclusion and Future works

6.1 Conclusion

The calculation of the Hessian matrix of a scalar function is a sub problem in many numerical optimization algorithms. For large-scale problems often the Hessian matrix is sparse and structured and it is preferable to exploit such information when available. Sometimes it is possible to determine from the problem structure the locations in the Hessian matrix that are identically zero. In this thesis,

- We have presented an efficient method to detect Hessian matrix sparsity pattern where the gradient is available as a black-box.
- On a set of large-scale practical test instances, we have observed a factor of 4 reduction (on average), in the number of gradient evaluations over the complete evaluation of the Hessian matrix (= the number of columns).

6.2 Future works

There are many opportunities for extending the scope of this thesis. This section presents some of these directions.

- For problems with special structures e.g., band matrices can be made very efficient because coloring and other relevant computations of band matrices will reduce the computational cost. Using band matrices instead of using randomly generated pattern at each level can be explored. This strategy is currently being implemented.

- A parallel implementation on shared memory symmetric multi-processor system is a promising research direction as there are ample opportunities for thread-level parallelism in the algorithms.

- The thesis uses the greedy CPR algorithm. More effective column partitioning consistent with direct determination allows structurally dependent column groups under certain restrictions. Methods in this category usually require fewer gradient evaluations compared with the CPR-type methods [34, 19].

- Voting scheme allows for the computation of Jacobian sparsity pattern detection. This is an exciting line of future research.

Bibliography

[1] Admat. `http://www.cayugaresearch.com`. Accessed: 2016-06-20.

[2] Bcs structural engineering matrices small test problem. `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcsstruc1/bcsstk01.html`. Accessed: 2016-06-20.

[3] The matrix market project. `http://math.nist.gov/MatrixMarket/`. Accessed: 2016-06-20.

[4] Power network pattern. `http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/bcspwr/bcspwr03_lg.html`. Accessed: 2016-06-20.

[5] The University of Florida Sparse Matrix Collection. `http://www.cise.u.edu/research/sparse/matrices/`. Accessed: 2016-06-20.

[6] Randal Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 281. Prentice Hall Upper Saddle River, 2003.

[7] Richard Geoffrey Carter. *Fast numerical determination of symmetric sparsity patterns*. Army High Performance Computing Research Center, 1992.

[8] Richard Geoffrey Carter, Shahadat Hossain, and Marzia Sultana. Efficient Detection of Hessian Matrix Sparsity Pattern. *Poster paper presented at 41st International Symposium on Symbolic and Algebraic Computation (ISSAC)*. Waterloo, 2016.

[9] Richard Geoffrey Carter, Shahadat Hossain, and Marzia Sultana. Efficient Detection of Hessian Matrix Sparsity Pattern. *To appear in ACM Communications in Computer Algebra (CCA)*, 2016.

[10] Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Software for estimating sparse Jacobian matrices. *ACM Transactions on Mathematical Software (TOMS)*, 10(3):329–345, 1984.

[11] Thomas F Coleman, Burton S Garbow, and Jorge J Moré. Software for estimating sparse Hessian matrices. *ACM Transactions on Mathematical Software (TOMS)*, 11(4):363–377, 1985.

[12] Thomas F Coleman and Jorge J Moré. Estimation of sparse Hessian matrices and graph coloring problems. *Mathematical Programming*, 28(3):243–270, 1984.

[13] 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, 1998.

[14] AR Curtis, Michael JD Powell, and John K Reid. On the estimation of sparse Jacobian matrices. *J. Inst. Math. Appl*, 13(1):117–120, 1974.

[15] John E Dennis Jr and Robert B Schnabel. *Numerical methods for unconstrained optimization and nonlinear equations*, volume 16. Siam, 1996.

[16] Shaun A Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Transactions on Mathematical Software (TOMS)*, 32(2):195–222, 2006.

[17] Assefaw H Gebremedhin, Arijit Tarafdar, Fredrik Manne, and Alex Pothen. New acyclic and star coloring algorithms with application to computing Hessians. *SIAM Journal on Scientific Computing*, 29(3):1042–1072, 2007.

[18] Assefaw H Gebremedhin, Arijit Tarafdar, Alex Pothen, and Andrea Walther. Efficient computation of sparse Hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, 21(2):209–223, 2009.

[19] Assefaw Hadish Gebremedhin, Fredrik Manne, and Alex Pothen. What color is your Jacobian? graph coloring for computing derivatives. *SIAM review*, 47(4):629–705, 2005.

[20] D Goldfarb and Ph L Toint. Optimal estimation of Jacobian and Hessian matrices that arise in finite difference calculations. *Mathematics of Computation*, 43(167):69–88, 1984.

[21] RM Gower and MP Mello. A new framework for the computation of Hessians. *Optimization Methods and Software*, 27(2):251–273, 2012.

[22] Andreas Griewank and Christo Mitev. Detecting Jacobian sparsity patterns by Bayesian probing. *Mathematical programming*, 93(1):1–25, 2002.

[23] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[24] Mahmudul Hasan, Shahadat Hossain, Ahamad Imtiaz Khan, Nasrin Hakim Mithila, and Ashraful Huq Suny. DSJM: a software toolkit for direct determination of sparse Jacobian matrices. In *Proceedings of the 5th International Conference on Mathematical Software, ICMS*, volume 9725 of *LNCS*, pages 275–283, Berlin,Germany, 2016. Springer.

[25] AKM Shahadat Hossain and Trond Steihaug. Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, 10(1):33–48, 1998.

[26] Shahadat Hossain and Trond Steihaug. Computing sparse Jacobian matrices optimally. In *Automatic Differentiation: Applications, Theory, and Implementations*, pages 77–87. Springer, 2006.

[27] Shahadat Hossain and Trond Steihaug. Graph models and their efficient implementation for sparse Jacobian matrix determination. *Discrete Applied Mathematics*, 161(12):1747–1754, 2013.

[28] Shahadat Hossain and Trond Steihaug. Optimal direct determination of sparse Jacobian matrices. *Optimization Methods and Software*, 28(6):1218–1232, 2013.

[29] Shahadat Hossain and Trond Steihaug. Sparse matrix computations with application to solve system of nonlinear equations. *Wiley Interdisciplinary Reviews: Computational Statistics*, 5(5):372–386, 2013.

[30] Shahadat Hossain and Marzia Sultana. Determination of Hessian Matrix Sparsity Pattern. *Poster paper presented at The Workshop on Nonlinear Optimization Algorithms and Industrial Applications*. The Fields Institute, Toronto, 2016.

[31] David Juedes and Jeffrey Jones. Coloring Jacobians revisited: a new algorithm for star and˜ acyclic bicoloring. *Optimization Methods and Software*, 27(2):295–309, 2012.

[32] S Thomas McCormick. Optimal approximation of sparse Hessians and its equivalence to a graph coloring problem. *Mathematical Programming*, 26(2):153–171, 1983.

[33] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[34] MJD Powell and Ph L Toint. On the estimation of sparse Hessian matrices. *SIAM Journal on Numerical Analysis*, 16(6):1060–1074, 1979.

[35] Andrea Walther. Computing sparse Hessians with automatic differentiation. *ACM Transactions on Mathematical Software (TOMS)*, 34(1):3, 2008.

[36] Andrea Walther. On the efficient computation of sparsity patterns for Hessians. In *Recent Advances in Algorithmic Differentiation*, pages 139–149. Springer, 2012.

[37] Andrea Walther and Andreas Griewank. Getting started with adol-c. In *Combinatorial scientific computing*, pages 181–202, 2009.

[38] JH Wilkinson. The algebraic eigenvalue problem. In *Handbook for Automatic Computation, Volume II, Linear Algebra*. Springer-Verlag New York, 1971.

[39] Wei Xu and Thomas F Coleman. Efficient (partial) determination of derivative matrices via automatic differentiation. *SIAM Journal on Scientific Computing*, 35(3):A1398–A1416, 2013.